Shell 编程范例



@泰晓科技 TinyLab.org

目錄

> 联系作者：笔者为一位重度开源践行者，可微信联系：lzufalcon
> 实验云台：在线学 Linux，Linux 0.11，汇编，Shell，C ...
> 本书来源：开源书籍：Shell 编程范例 (by 泰晓科技)
> 报名参与：*Star/fork* GitHub 仓库 并发送 *Pull Request*
> 关注我们：扫描二维码 关注 @泰晓科技 微博和微信公众号
> 赞助我们：赞助 8.99￥，更多原创开源书籍期待您的支持 ^o^

Shell 编程范例

v 0.3

不同于传统 Shell 书籍，本书并未花大篇幅去介绍 Shell 语法，而是以面向"对象" 的方式引入大量的实例介绍 Shell 日常操作，"对象" 涵盖数值、逻辑值、字符串、文件、进程、文件系统等。这样有助于学以致用，并在用的过程中提高兴趣。也可以作为 Shell 编程索引，在需要的时候随时检索。

介绍

- 项目首页：http://www.tinylab.org/open-shell-book
- 代码仓库：https://github.com/tinyclub/open-shell-book
- 在线阅读：http://tinylab.gitbooks.io/shellbook
- 实验云台：在线学 Linux，Linux 0.11，汇编，Shell，C ...

  更多背景和计划请参考：前言。

编译

要编译本书，请使用 Markdown Lab。

纠错

欢迎大家指出不足，如有任何疑问，请邮件联系 wuzhangjin at gmail dot com 或者直接修复并提交 Pull Request。

版权

关注我们

- 新浪微博



- 微信公众号



赞助我们

- 微信扫码赞助原创



- 访问 泰晓开源小店 支持心仪项目



更多原创开源书籍

- C 语言编程透视
- 嵌入式 Linux 知识库(eLinux.org 中文版)
- Linux 内核文档(Linux Documentation/ 中文版)

版本修订历史

Revision	Author	From	Date	Description
0.3	@吴章金falcon	@泰晓科技	2016/08/30	增加一章: 用户管理
0.2	@吴章金falcon	@泰晓科技	2015/07/23	调整格式，修复链接
0.1	@吴章金falcon	@泰晓科技	2014/01/07	初稿

前言

- 背景
- 现状
- 计划

背景

早在 2007 年 11 月，为了系统地学习和总结 Shell 编程，作者专门制定了一个 Shell 编程范例的总结计划，当时的计划是：

> 这个系列将以面向“对象”（即我们操作的对象）来展开，并引入大量的实例，这样有助于让我们真正去学以致用，并在用的过程中提高兴趣。所以这个系列将不会专门介绍 Shell 的语法， 而是假设读者对 Shell 编程有了一定的基础。
>
> 另外，该系列到最后可能会涵盖：数值、逻辑值、字符串、文件、进程、文件系统等所有我们可以操作的“对象”，这个操作对象也将从低级到高级，进而上升到网络层面，整个通过各种方式连接起来的计算机的集合。实际上这也未尝不是在摸索 UNIX 的哲学，那"K.I.S.S"（Keep It Simple, Stupid）蕴藏的巨大能量。
>
> —— 摘自《兰大开源社区 >> 脚本编程 >> Shell 编程范例》

2008 年 4 月底，整个系列大部分内容和框架基本完成，后来由于实习和工作原因，并没有持续完善。不过相关章节却获得了较好的反响，很多热心网友有大量评论和转载，例如，在百度文库转载的一份《Shell编程范例之字符串操作》的访问量已接近 3000。说明整个系列还是有比较大的阅读群体。

现状

考虑到整个 Linux 世界的蓬勃发展，Shell 的使用环境越来越多，相关使用群体会不断增加，所以最近已经将该系列重新整理，并以自由书籍的方式发布，以便惠及更多的读者。

整个系列已经用 Markdown 重新组织，并发布到了 泰晓科技|TinyLab.org。

整理到TinyLab.org的索引篇是：《Shell编程范例之索引篇》，其内容结构如下：

- Shell编程范例之开篇 (更新时间：2007-07-21)
- Shell编程范例之数值运算 (更新时间：2007-11-9)
- Shell编程范例之布尔运算 (更新时间：2007-10-30)
- Shell编程范例之字符串操作 (更新时间：2007-11-21)

- Shell编程范例之文件操作 (更新时间：2007-12-5)
- Shell编程范例之文件系统操作 (更新时间：2007-12-29)
- Shell编程范例之进程操作 (更新时间：2008-02-22)
- Shell编程范例之网络操作 (更新时间：2008-04-19)
- Shell编程范例之总结篇 (更新时间：2008-07-21)

最近，基于一个 Markdown 的开源书籍模版：Gitbook，已经把该系列整理成了自由书籍，并维护在 TinyLab 的项目仓库中。项目相关信息如下：

- 项目首页：http://www.tinylab.org/open-shell-book/
- 代码仓库：https://github.com/tinyclub/open-shell-book.git

计划

后续除了继续在 泰晓科技|TinyLab.org 以 Blog 形式持续更新外，还打算重新规划、增补整个系列，并以自由书籍的方式持续维护，并通过 TinLab.org 平台接受读者的反馈，直到正式发行出版。

欢迎大家指出本书初稿中的不足，甚至参与到相关章节的写作、校订和完善中来。

如果有时间和兴趣，欢迎参与。可以通过 泰晓科技 联系我们，或者直接关注微博@泰晓科技并私信我们。

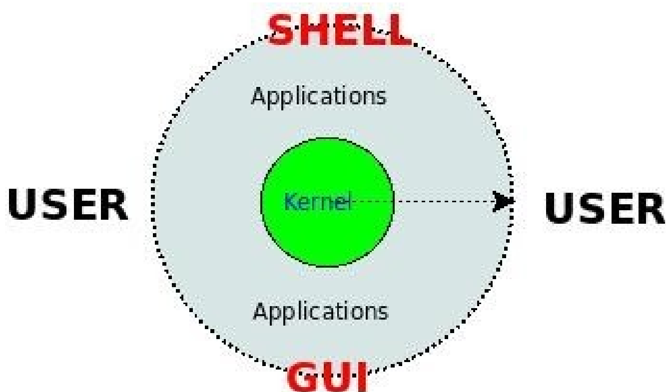
准备工作

前言

到最后一节来写“开篇”，确实有点古怪。不过，在第一篇（数值操作）的开头实际上也算是一个小的开篇，那里提到整个系列的前提是需要有一定的 Shell 编程基础，因此，为了能够让没有 Shell 编程基础的读者也可以阅读这个系列，我到最后来重写这个开篇。开篇主要介绍什么是 Shell，Shell 运行环境，Shell 基本语法和调试技巧。

什么是 Shell

首先让我们从下图看看 Shell 在整个操作系统中所处的位置吧，该图的外圆描述了整个操作系统（比如 `Debian/Ubuntu/Slackware` 等），内圆描述了操作系统的核心（比如 `Linux Kernel` ），而 `Shell` 和 `GUI` 一样作为用户和操作系统之间的接口。



`GUI` 提供了一种图形化的用户接口，使用起来非常简便易学；而 `Shell` 则为用户提供了一种命令行的接口，接收用户的键盘输入，并分析和执行输入字符串中的命令，然后给用户返回执行结果，使用起来可能会复杂一些，但是由于占用的资源少，而且在操作熟练以后可能会提高工作效率，而且具有批处理的功能，因此在某些应用场合还非常流行。

`Shell` 作为一种用户接口，它实际上是一个能够解释和分析用户键盘输入，执行输入中的命令，然后返回结果的一个解释程序（Interpreter，例如在 `linux` 下比较常用的 `Bash` ），我们可以通过下面的命令查看当前的 `Shell` ：

```
$ echo $SHELL
/bin/bash
$ ls -l /bin/bash
-rwxr-xr-x 1 root root 702160 2008-05-13 02:33 /bin/bash
```

该解释程序不仅能够解释简单的命令，而且可以解释一个具有特定语法结构的文件，这种文件被称作脚本（Script）。它具体是如何解释这些命令和脚本文件的，这里不深入分析，请看我在 2008 年写的另外一篇文章：《Linux命令行上程序执行的一刹那》。

既然该程序可以解释具有一定语法结构的文件，那么我们就可以遵循某一语法来编写它，它有什么样的语法，如何运行，如何调试呢？下面我们以 `Bash` 为例来讨论这几个方面。

搭建运行环境

为了方便后面的练习，我们先搭建一个基本运行环境：在一个 Linux 操作系统中，有一个运行有 `Bash` 的命令行在等待我们键入命令，这个命令行可以是图形界面下的 `Terminal` （例如 `Ubuntu` 下非常厉害的 `Terminator` ），也可以是字符界面的 `Console` （可以用 `CTRL+ALT+F1~6` 切换），如果你发现当前 `Shell` 不是 `Bash`，请用下面的方法替换它：

```
$ chsh $USER -s /bin/bash
$ su $USER
```

或者是简单地键入Bash：

```
$ bash
$ echo $SHELL  # 确认一下
/bin/bash
```

如果没有安装 Linux 操作系统，也可以考虑使用一些公共社区提供的 Linux 虚拟实验服务，一般都有提供远程 `Shell`，你可以通过 `Telnet` 或者是 `Ssh` 的客户端登录上去进行练习。

有了基本的运行环境，那么如何来运行用户键入的命令或者是用户编写好的脚本文件呢 `?`

假设我们编写好了一个 Shell 脚本，叫 `test.sh` 。

第一种方法是确保我们执行的命令具有可执行权限，然后直接键入该命令执行它：

```
$ chmod +x /path/to/test.sh
$ /path/to/test.sh
```

第二种方法是直接把脚本作为 `Bash` 解释器的参数传入：

```
$ bash /path/to/test.sh
```

或

```
$ source /path/to/test.sh
```

或

```
$ . /path/to/test.sh
```

基本语法介绍

先来一个 `Hello, World` 程序。

下面来介绍一个 Shell 程序的基本结构，以 `Hello, World` 为例：

```
#!/bin/bash -v
# test.sh
echo "Hello, World"
```

把上述代码保存为 `test.sh`，然后通过上面两种不同方式运行，可以看到如下效果。

方法一：

```
$ chmod +x test.sh
$ ./test.sh
 ./test.sh
 #!/bin/bash -v

 echo "Hello, World"
 Hello, World
```

方法二：

```
$ bash test.sh
Hello, World

$ source test.sh
Hello, World

$ . test.sh
Hello, World
```

我们发现两者运行结果有区别，为什么呢？这里我们需要关注一下 `test.sh` 文件的内容，它仅仅有两行，第二行打印了 `Hello, World` ，两种方法都达到了目的，但是第一种方法却多打印了脚本文件本身的内容，为什么呢？

原因在该文件的第一行，当我们直接运行该脚本文件时，该行告诉操作系统使用用 `#!` 符号之后面的解释器以及相应的参数来解释该脚本文件，通过分析第一行，我们发现对应的解释器以及参数是 `/bin/bash -v` ，而 `-v` 刚好就是要打印程序的源代码；但是我们在用第二种方法时没有给 `Bash` 传递任何额外的参数，因此，它仅仅解释了脚本文件本身。

其他语法细节请直接看《Shell编程学习笔记》即本书后面的附录一。

Shell 程序设计过程

Shell 语言作为解释型语言，它的程序设计过程跟编译型语言有些区别，其基本过程如下：

- 设计算法
- 用 Shell 编写脚本程序实现算法
- 直接运行脚本程序

可见它没有编译型语言的"麻烦的"编译和链接过程，不过正是因为这样，它出错时调试起来不是很方便，因为语法错误和逻辑错误都在运行时出现。下面我们简单介绍一下调试方法。

调试方法介绍

可以直接参考资料：Shell 脚本调试技术 或者 BASH 的调试手段。

小结

Shell 语言作为一门解释型语言，可以使用大量的现有工具，包括数值计算、符号处理、文件操作、网络操作等，因此，编写过程可能更加高效，但是因为它是解释型的，需要在执行过程中从磁盘上不断调用外部的程序并进行进程之间的切换，在运行效率方面可能有劣势，所以我们应该根据应用场合选择使用 Shell 或是用其他的语言来编程。

参考资料

- Linux命令行上程序执行的一刹那
- Linux Shell编程学习笔记
- Shell 脚本调试技术
- BASH 的调试手段

数值运算

前言

从本文开始，打算结合平时积累和进一步实践，通过一些范例来介绍Shell编程。因为范例往往能够给人以学有所用的感觉，而且给人以动手实践的机会，从而激发人的学习热情。

考虑到易读性，这些范例将非常简单，但是实用，希望它们能够成为我们解决日常问题的参照物或者是“茶余饭后”的小点心，当然这些“点心”肯定还有值得探讨、优化的地方。

更复杂有趣的例子请参考 Advanced Bash-Scripting Guide (一本深入学习 Shell 脚本艺术的书籍)。

该系列概要：

- 目的：享受用 Shell 解决问题的乐趣；和朋友们一起交流和探讨。

- 计划：先零散地写些东西，之后再不断补充，最后整理成册。
- 读者：熟悉 Linux 基本知识，如文件系统结构、常用命令行工具、Shell 编程基础等。
- 建议：看范例时，可参考《Shell基础十二篇》和《Shell十三问》。
- 环境：如没特别说明，该系列使用的 Shell 将特指 Bash，版本在 3.1.17 以上。
- 说明：该系列不是依据 Shell 语法组织，而是面向某些潜在的操作对象和操作本身，它们反应了现实应用。当然，在这个过程中肯定会涉及到 Shell 的语法。

这一篇打算讨论一下 Shell 编程中的基本数值运算，这类运算包括：

- 数值（包括整数和浮点数）间的加、减、乘、除、求幂、求模等
- 产生指定范围的随机数
- 产生指定范围的数列

Shell 本身可以做整数运算，复杂一些的运算要通过外部命令实现，比如 `expr`，`bc`，`awk` 等。另外，可通过 `RANDOM` 环境变量产生一个从 0 到 32767 的随机数，一些外部工具，比如 `awk` 可以通过 `rand()` 函数产生随机数。而 `seq` 命令可以用来产生一个数列。下面对它们分别进行介绍。

整数运算

范例：对某个数加 1

```
$ i=0;
$ ((i++))
$ echo $i
1

$ let i++
$ echo $i
2

$ expr $i + 1
3
$ echo $i
2

$ echo $i 1 | awk '{printf $1+$2}'
3
```

说明：`expr` 之后的 `$i`，`+`，1 之间有空格分开。如果进行乘法运算，需要对运算符进行转义，否则 Shell 会把乘号解释为通配符，导致语法错误；`awk` 后面的 `$1` 和 `$2` 分别指 `$i` 和 1，即从左往右的第 1 个和第 2 个数。

用 Shell 的内置命令查看各个命令的类型如下：

```
$ type type
type is a shell builtin
$ type let
let is a shell builtin
$ type expr
expr is hashed (/usr/bin/expr)
$ type bc
bc is hashed (/usr/bin/bc)
$ type awk
awk is /usr/bin/awk
```

从上述演示可看出： `let` 是 Shell 内置命令，其他几个是外部命令，都在 `/usr/bin` 目录下。而 `expr` 和 `bc` 因为刚用过，已经加载在内存的 `hash` 表中。这将有利于我们理解在上一章介绍的脚本多种执行方法背后的原理。

说明：如果要查看不同命令的帮助，对于 `let` 和 `type` 等 Shell 内置命令，可以通过 Shell 的一个内置命令 `help` 来查看相关帮助，而一些外部命令可以通过 Shell 的一个外部命令 `man` 来查看帮助，用法诸如 `help let` ， `man expr` 等。

范例：从 1 加到某个数

```
#!/bin/bash
# calc.sh

i=0;
while [ $i -lt 10000 ]
do
    ((i++))
done
echo $i
```

说明：这里通过 `while [ 条件表达式 ]; do .... done` 循环来实现。 `-lt` 是小于号 `<` ，具体见 `test` 命令的用法： `man test` 。

如何执行该脚本？

办法一：直接把脚本文件当成子 Shell （Bash）的一个参数传入

```
$ bash calc.sh
$ type bash
bash is hashed (/bin/bash)
```

办法二：是通过 `bash` 的内置命令 `.` 或 `source` 执行

```
$ . ./calc.sh
```

或

```
$ source ./calc.sh
$ type .
. is a shell builtin
$ type source
source is a shell builtin
```

办法三：是修改文件为可执行，直接在当前 Shell 下执行

```
$ chmod ./calc.sh
$ ./calc.sh
```

下面，逐一演示用其他方法计算变量加一，即把 `((i++))` 行替换成下面的某一个：

```
let i++;

i=$(expr $i + 1)

i=$(echo $i+1|bc)

i=$(echo "$i 1" | awk '{printf $1+$2;}')
```

比较计算时间如下：

```
$ time calc.sh
10000

real    0m1.319s
user    0m1.056s
sys     0m0.036s
$ time calc_let.sh
10000

real    0m1.426s
user    0m1.176s
sys     0m0.032s
$  time calc_expr.sh
1000

real    0m27.425s
user    0m5.060s
sys     0m14.177s
$ time calc_bc.sh
1000

real    0m56.576s
user    0m9.353s
sys     0m24.618s
$ time ./calc_awk.sh
100

real    0m11.672s
user    0m2.604s
sys     0m2.660s
```

说明： `time` 命令可以用来统计命令执行时间，这部分时间包括总的运行时间，用户空间执行时间，内核空间执行时间，它通过 `ptrace` 系统调用实现。

通过上述比较可以发现 `(())` 的运算效率最高。而 `let` 作为 Shell 内置命令，效率也很高，但是 `expr` ， `bc` ， `awk` 的计算效率就比较低。所以，在 Shell 本身能够完成相关工作的情况下，建议优先使用 Shell 本身提供的功能。但是 Shell 本身无法完成的功能，比如浮点运算，所以就需要外部命令的帮助。另外，考虑到 Shell 脚本的可移植性，在性能不是很关键的情况下，不要使用某些 Shell 特有的语法。

`let` ， `expr` ， `bc` 都可以用来求模，运算符都是 `%` ，而 `let` 和 `bc` 可以用来求幂，运算符不一样，前者是 `**` ，后者是 `^` 。例如：

范例：求模

```
$ expr 5 % 2
1

$ let i=5%2
$ echo $i
1

$ echo 5 % 2 | bc
1

$ ((i=5%2))
$ echo $i
1
```

范例：求幂

```
$ let i=5**2
$ echo $i
25

$ ((i=5**2))
$ echo $i

25
$ echo "5^2" | bc
25
```

范例：进制转换

进制转换也是比较常用的操作，可以用 `Bash` 的内置支持也可以用 `bc` 来完成，例如把 8 进制的 11 转换为 10 进制，则可以：

```
$ echo "obase=10;ibase=8;11" | bc -l
9

$ echo $((8#11))
9
```

上面都是把某个进制的数转换为 10 进制的，如果要进行任意进制之间的转换还是 `bc` 比较灵活，因为它可以直接用 `ibase` 和 `obase` 分别指定进制源和进制转换目标。

范例：ascii 字符编码

如果要把某些字符串以特定的进制表示，可以用 `od` 命令，例如默认的分隔符 `IFS` 包括空格、 `TAB` 以及换行，可以用 `man ascii` 佐证。

```
$ echo -n "$IFS" | od -c
0000000      t  n
0000003
$ echo -n "$IFS" | od -b
0000000 040 011 012
0000003
```

浮点运算

`let` 和 `expr` 都无法进行浮点运算，但是 `bc` 和 `awk` 可以。

范例：求 1 除以 13，保留 3 位有效数字

```
$ echo "scale=3; 1/13"  | bc
.076

$ echo "1 13" | awk '{printf("%0.3f\n",$1/$2)}'
0.077
```

说明： `bc` 在进行浮点运算时需指定精度，否则默认为 0，即进行浮点运算时，默认结果只保留整数。而 `awk` 在控制小数位数时非常灵活，仅仅通过 `printf` 的格式控制就可以实现。

补充：在用 `bc` 进行运算时，如果不用 `scale` 指定精度，而在 `bc` 后加上 `-l` 选项，也可以进行浮点运算，只不过这时的默认精度是 20 位。例如：

```
$ echo 1/13100 | bc -l
.00007633587786259541
```

范例：余弦值转角度

用 `bc -l` 计算，可以获得高精度：

```
$ export cos=0.996293; echo "scale=100; a(sqrt(1-$cos^2)/$cos)*180/(a(1)*4)" | bc -l
4.934954755411383632719834036931840605159706398655243875372764917732
5495504159766011527078286004072131
```

当然也可以用 `awk` 来计算：

```
$ echo 0.996293 | awk '{ printf("%s\n", atan2(sqrt(1-$1^2),$1)*180/3.1415926535);}'
4.93495
```

范例：有一组数据，求人均月收入最高家庭

在这里随机产生了一组测试数据，文件名为 `income.txt`。

```
1 3 4490
2 5 3896
3 4 3112
4 4 4716
5 4 4578
6 6 5399
7 3 5089
8 6 3029
9 4 6195
10 5 5145
```

说明：上面的三列数据分别是家庭编号、家庭人数、家庭月总收入。

分析：为了求月均收入最高家庭，需要对后面两列数进行除法运算，即求出每个家庭的月均收入，然后按照月均收入排序，找出收入最高家庭。

实现：

```
#!/bin/bash
# gettopfamily.sh

[ $# -lt 1 ] && echo "please input the income file" && exit -1
[ ! -f $1 ] && echo "$1 is not a file" && exit -1

income=$1
awk '{
    printf("%d %0.2f\n", $1, $3/$2);
}' $income | sort -k 2 -n -r
```

说明：

- `[ $# -lt 1 ]`：要求至少输入一个参数，`$#` 是 Shell 中传入参数的个数
- `[ ! -f $1 ]`：要求输入参数是一个文件，`-f` 的用法见 `test` 命令，`man test`
- `income=$1`：把输入参数赋给 income 变量，再作为 `awk` 的参数，即需处理的文件
- `awk`：用文件第三列除以第二列，求出月均收入，考虑到精确性，保留了两位精度
- `sort -k 2 -n -r`：这里对结果的 `awk` 结果的第二列 `-k 2`，即月均收入进行排序，按照数字排序 `-n`，并按照递减的顺序排序 `-r`。

演示：

```
$ ./gettopfamily.sh income.txt
7 1696.33
9 1548.75
1 1496.67
4 1179.00
5 1144.50
10 1029.00
6 899.83
2 779.20
3 778.00
8 504.83
```

补充：之前的 `income.txt` 数据是随机产生的。在做一些实验时，往往需要随机产生一些数据，在下一小节，我们将详细介绍它。这里是产生 `income.txt` 数据的脚本：

```
#!/bin/bash
# genrandomdata.sh

for i in $(seq 1 10)
do
    echo $i $(($RANDOM/8192+3)) $((RANDOM/10+3000))
done
```

说明：上述脚本中还用到 `seq` 命令产生从1到10的一列数，这个命令的详细用法在该篇最后一节也会进一步介绍。

随机数

环境变量 `RANDOM` 产生从 0 到 32767 的随机数，而 `awk` 的 `rand()` 函数可以产生 0 到 1 之间的随机数。

范例：获取一个随机数

```
$ echo $RANDOM
81

$ echo "" | awk '{srand(); printf("%f", rand());}'
0.237788
```

说明： `srand()` 在无参数时，采用当前时间作为 `rand()` 随机数产生器的一个 `seed` 。

范例：随机产生一个从 **0** 到 **255** 之间的数字

可以通过 `RANDOM` 变量的缩放和 `awk` 中 `rand()` 的放大来实现。

```
$ expr $RANDOM / 128

$ echo "" | awk '{srand(); printf("%d\n", rand()*255);}'
```

思考：如果要随机产生某个 IP 段的 IP 地址，该如何做呢？看例子：友善地获取一个可用的 IP 地址。

```
#!/bin/bash
# getip.sh -- get an usable ipaddress automatically
# author: falcon &lt;zhangjinw@gmail.com>
# update: Tue Oct 30 23:46:17 CST 2007

# set your own network, default gateway, and the time out of "ping" command
net="192.168.1"
default_gateway="192.168.1.1"
over_time=2

# check the current ipaddress
ping -c 1 $default_gateway -W $over_time
[ $? -eq 0 ] && echo "the current ipaddress is okey!" && exit -1;

while :; do
    # clear the current configuration
    ifconfig eth0 down
    # configure the ip address of the eth0
    ifconfig eth0 \
        $net.$(($RANDOM /130 +2)) \
        up
    # configure the default gateway
    route add default gw $default_gateway
    # check the new configuration
    ping -c 1 $default_gateway -W $over_time
    # if work, finish
    [ $? -eq 0 ] && break
done
```

说明：如果你的默认网关地址不是 `192.168.1.1`，请自行配置 `default_gateway`（可以用 `route -n` 命令查看），因为用 `ifconfig` 配置地址时不能配置为网关地址，否则你的IP地址将和网关一样，导致整个网络不能正常工作。

其他运算

其实通过一个循环就可以产生一系列数，但是有相关工具为什么不用呢！`seq` 就是这么一个小工具，它可以产生一系列数，你可以指定数的递增间隔，也可以指定相邻两个数之间的分割符。

范例：获取一系列数

```
$ seq 5
1
2
3
4
5
$ seq 1 5
1
2
3
4
5
$ seq 1 2 5
1
3
5
$ seq -s: 1 2 5
1:3:5
$ seq 1 2 14
1
3
5
7
9
11
13
$ seq -w 1 2 14
01
03
05
07
09
11
13
$ seq -s: -w 1 2 14
01:03:05:07:09:11:13
$ seq -f "0x%g" 1 5
0x1
0x2
0x3
0x4
0x5
```

一个比较典型的使用 `seq` 的例子，构造一些特定格式的链接，然后用 `wget` 下载这些内容：

```
$ for i in `seq -f"http://thns.tsinghua.edu.cn/thnsebooks/ebook73/%02g.pdf" 1 21`;do w
get -c $i; done
```

或者

```
$ for i in `seq -w 1 21`;do wget -c "http://thns.tsinghua.edu.cn/thnsebooks/ebook73/$i
"; done
```

补充：在 `Bash` 版本 3 以上，在 `for` 循环的 `in` 后面，可以直接通过 `{1..5}` 更简洁地产生自 1 到 5 的数字（注意，1 和 5 之间只有两个点），例如：

```
$ for i in {1..5}; do echo -n "$i "; done
1 2 3 4 5
```

范例：统计字符串中各单词出现次数

我们先给单词一个定义：由字母组成的单个或者多个字符系列。

首先，统计每个单词出现的次数：

```
$ wget -c http://tinylab.org
$ cat index.html | sed -e "s/[^a-zA-Z]/\n/g" | grep -v ^$ | sort | uniq -c
```

接着，统计出现频率最高的前10个单词：

```
$ wget -c http://tinylab.org
$ cat index.html | sed -e "s/[^a-zA-Z]/\n/g" | grep -v ^$ | sort | uniq -c | sort -n -
k 1 -r | head -10
    524 a
    238 tag
    205 href
    201 class
    193 http
    189 org
    175 tinylab
    174 www
    146 div
    128 title
```

说明：

- `cat index.html`：输出 index.html 文件里的内容
- `sed -e "s/[^a-zA-Z]/\n/g"`：把非字母字符替换成空格，只保留字母字符
- `grep -v ^$`：去掉空行
- `sort`：排序
- `uniq -c`：统计相同行的个数，即每个单词的个数
- `sort -n -k 1 -r`：按照第一列 `-k 1` 的数字 `-n` 逆序 `-r` 排序
- `head -10`：取出前十行

范例：统计指定单词出现次数

可以考虑采取两种办法：

- 只统计那些需要统计的单词
- 用上面的算法把所有单词的个数都统计出来，然后再返回那些需要统计的单词给用户

不过，这两种办法都可以通过下面的结构来实现。先看办法一：

```
#!/bin/bash
# statistic_words.sh

if [ $# -lt 1 ]; then
    echo "Usage: basename $0 FILE WORDS ...."
    exit -1
fi

FILE=$1
((WORDS_NUM=$#-1))

for n in $(seq $WORDS_NUM)
do
    shift
    cat $FILE | sed -e "s/[^a-zA-Z]/\n/g" \
        | grep -v ^$ | sort | grep ^$1$ | uniq -c
done
```

说明：

- `if 条件部分`：要求至少两个参数，第一个单词文件，之后参数为要统计的单词
- `FILE=$1`：获取文件名，即脚本之后的第一个字符串
- `((WORDS_NUM=$#-1))`：获取单词个数，即总的参数个数 `$#` 减去文件名参数（1个）
- `for 循环部分`：首先通过 `seq` 产生需要统计的单词个数系列，`shift` 是 Shell 内置变量（请通过 `help shift` 获取帮助)，它把用户从命令行中传入的参数依次往后移动位置，并把当前参数作为第一个参数即 `$1`，这样通过 `$1` 就可以遍历用户所有输入的单词（仔细一想，这里貌似有数组下标的味道）。你可以考虑把 `shift` 之后的那句替换成 `echo $1` 测试 `shift` 的用法

演示：

```
$ chmod +x statistic_words.sh
$ ./statistic_words.sh index.html tinylab linux python
    175 tinylab
     43 linux
      3 python
```

再看办法二，我们只需要修改 `shift` 之后的那句即可：

```
#!/bin/bash
# statistic_words.sh

if [ $# -lt 1 ]; then
    echo "ERROR: you should input 2 words at least";
    echo "Usage: basename $0 FILE WORDS ...."
    exit -1
fi

FILE=$1
((WORDS_NUM=$#-1))

for n in $(seq $WORDS_NUM)
do
    shift
    cat $FILE | sed -e "s/[^a-zA-Z]/\n/g" \
        | grep -v ^$ | sort | uniq -c | grep " $1$"
done
```

演示：

```
$ ./statistic_words.sh index.html tinylab linux python
    175 tinylab
     43 linux
      3 python
```

说明：很明显，办法一的效率要高很多，因为它提前找出了需要统计的单词，然后再统计，而后者则不然。实际上，如果使用 `grep` 的 `-E` 选项，我们无须引入循环，而用一条命令就可以搞定：

```
$ cat index.html | sed -e "s/[^a-zA-Z]/\n/g" | grep -v ^$ | sort | grep -E "^tinylab$|
^linux$" | uniq -c
     43 linux
    175 tinylab
```

或者

```
$ cat index.html | sed -e "s/[^a-zA-Z]/\n/g" | grep -v ^$ | sort | egrep  "^tinylab$|^
linux$" | uniq -c
     43 linux
    175 tinylab
```

说明：需要注意到 `sed` 命令可以直接处理文件，而无需通过 `cat` 命令输出以后再通过管道传递，这样可以减少一个不必要的管道操作，所以上述命令可以简化为：

```
$ sed -e "s/[^a-zA-Z]/\n/g" index.html | grep -v ^$ | sort | egrep  "^tinylab$|^linux$
" | uniq -c
     43 linux
    175 tinylab
```

所以，可见这些命令 `sed`，`grep`，`uniq`，`sort` 是多么有用，它们本身虽然只完成简单的功能，但是通过一定的组合，就可以实现各种五花八门的事情啦。对了，统计单词还有个非常有用的命令 `wc -w`，需要用到的时候也可以用它。

补充：在 Advanced Bash-Scripting Guide 一书中还提到 `jot` 命令和 `factor` 命令，由于机器上没有，所以没有测试，`factor` 命令可以产生某个数的所有素数。如：

```
$ factor 100
100: 2 2 5 5
```

小结

到这里，Shell 编程范例之数值计算就结束啦。该篇主要介绍了：

- Shell 编程中的整数运算、浮点运算、随机数的产生、数列的产生
- Shell 的内置命令、外部命令的区别，以及如何查看他们的类型和帮助
- Shell 脚本的几种执行办法
- 几个常用的 Shell 外部命令：`sed`，`awk`，`grep`，`uniq`，`sort` 等
- 范例：数字递增；求月均收入；自动获取 `IP` 地址；统计单词个数
- 其他：相关用法如命令列表，条件测试等在上述范例中都已涉及，请认真阅读之

如果您有时间，请温习之。

资料

- Advanced Bash-Scripting Guide
- shell 十三问
- shell 基础十二篇

- SED 手册
- AWK 使用手册
- 几个 Shell 讨论区
    - LinuxSir.org
    - ChinaUnix.net

后记

大概花了 3 个多小时才写完，目前是 23:33，该回宿舍睡觉啦，明天起来修改错别字和补充一些内容，朋友们晚安！

10 月 31 号，修改部分措辞，增加一篇统计家庭月均收入的范例，添加总结和参考资料，并用附录所有代码。

Shell 编程是一件非常有趣的事情，如果您想一想：上面计算家庭月均收入的例子，然后和用 `M$ Excel` 来做这个工作比较，你会发现前者是那么简单和省事，而且给您以运用自如的感觉。

布尔运算

前言

上个礼拜介绍了Shell编程范例之数值运算，对 Shell 下基本数值运算方法做了简单的介绍，这周将一起探讨布尔运算，即如何操作“真假值”。

在 Bash 里有这样的常量(实际上是两个内置命令，在这里我们姑且这么认为，后面将介绍)，即 true 和 false，一个表示真，一个表示假。对它们可以进行与、或、非运算等常规的逻辑运算，在这一节，我们除了讨论这些基本逻辑运算外，还将讨论Shell编程中的条件测试和命令列表，并介绍它们和布尔运算的关系。

常规的布尔运算

这里主要介绍 `Bash` 里头常规的逻辑运算，与、或、非。

在 Shell 下如何进行逻辑运算

范例：true or false

单独测试 `true` 和 `false` ，可以看出 `true` 是真值， `false` 为假

```
$ if true;then echo "YES"; else echo "NO"; fi
YES
$ if false;then echo "YES"; else echo "NO"; fi
NO
```

范例：与运算

```
$ if true && true;then echo "YES"; else echo "NO"; fi
YES
$ if true && false;then echo "YES"; else echo "NO"; fi
NO
$ if false && false;then echo "YES"; else echo "NO"; fi
NO
$ if false && true;then echo "YES"; else echo "NO"; fi
NO
```

范例：或运算

```
$ if true || true;then echo "YES"; else echo "NO"; fi
YES
$ if true || false;then echo "YES"; else echo "NO"; fi
YES
$ if false || true;then echo "YES"; else echo "NO"; fi
YES
$ if false || false;then echo "YES"; else echo "NO"; fi
NO
```

范例：非运算，即取反

```
$ if ! false;then echo "YES"; else echo "NO"; fi
YES
$ if ! true;then echo "YES"; else echo "NO"; fi
NO
```

可以看出 `true` 和 `false` 按照我们对逻辑运算的理解进行着，但是为了能够更好的理解 Shell 对逻辑运算的实现，我们还得弄清楚，`true` 和 `false` 是怎么工作的？

Bash 里头的 true 和 false 是我们通常认为的 1 和 0 么？

回答是：否。

范例：返回值 v.s. 逻辑值

`true` 和 `false` 它们本身并非逻辑值，它们都是 Shell 的内置命令，只是它们的返回值是一个“逻辑值”：

```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

可以看到 `true` 返回了 0，而 `false` 则返回了 1 。跟我们离散数学里学的真值 1 和 0 并不是对应的，而且相反的。

范例：查看 true 和 false 帮助和类型

```
$ help true false
true: true
     Return a successful result.
false: false
     Return an unsuccessful result.
$ type true false
true is a shell builtin
false is a shell builtin
```

说明：`$?` 是一个特殊变量，存放有上一次进程的结束状态（退出状态码）。

从上面的操作不难联想到在 C 语言程序设计中为什么会强调在 `main` 函数前面加上 `int`，并在末尾加上 `return 0`。因为在 Shell 里，将把 0 作为程序是否成功结束的标志，这就是 Shell 里头 `true` 和 `false` 的实质，它们用以反应某个程序是否正确结束，而并非传统的真假值（1 和 0），相反地，它们返回的是 0 和 1 。不过庆幸地是，我们在做逻辑运算时，无须关心这些。

条件测试

从上节中，我们已经清楚地了解了 Shell 下的“逻辑值”是什么：是进程退出时的返回值，如果成功返回，则为真，如果不成功返回，则为假。

而条件测试正好使用了 `test` 这么一个指令，它用来进行数值测试（各种数值属性测试）、字符串测试（各种字符串属性测试）、文件测试（各种文件属性测试），我们通过判断对应的测试是否成功，从而完成各种常规工作，再加上各种测试的逻辑组合后，将可以完成更复杂的工作。

条件测试基本使用

范例：数值测试

```
$ if test 5 -eq 5;then echo "YES"; else echo "NO"; fi
YES
$ if test 5 -ne 5;then echo "YES"; else echo "NO"; fi
NO
```

范例：字符串测试

```
$ if test -n "not empty";then echo "YES"; else echo "NO"; fi
YES
$ if test -z "not empty";then echo "YES"; else echo "NO"; fi
NO
$ if test -z "";then echo "YES"; else echo "NO"; fi
YES
$ if test -n "";then echo "YES"; else echo "NO"; fi
NO
```

范例：文件测试

```
$ if test -f /boot/System.map; then echo "YES"; else echo "NO"; fi
YES
$ if test -d /boot/System.map; then echo "YES"; else echo "NO"; fi
NO
```

各种逻辑测试的组合

范例：如果 a，b，c 都等于下面对应的值，那么打印 YES，通过 -a 进行"与"测试

```
$ a=5;b=4;c=6;
$ if test $a -eq 5 -a $b -eq 4 -a $c -eq 6; then echo "YES"; else echo "NO"; fi
YES
```

范例：测试某个“东西”是文件或者目录，通过 -o 进行“或”运算

```
$ if test -f /etc/profile -o -d /etc/profile;then echo "YES"; else echo "NO"; fi
YES
```

范例：测试某个“东西”是否为文件，测试 `!` 非运算

```
$ if test ! -f /etc/profile; then echo "YES"; else echo "NO"; fi
NO
```

上面仅仅演示了 `test` 命令一些非常简单的测试，你可以通过 `help test` 获取 `test` 的更多用法。需要注意的是，`test` 命令内部的逻辑运算和 Shell 的逻辑运算符有一些区别，对应的为 `-a` 和 `&&`，`-o` 与 `||`，这两者不能混淆使用。而非运算都是 `!`，下面对它们进行比较。

比较 -a 与 &&, -o 与 ||，! test 与 test !

范例：要求某文件可执行且有内容，用 -a 和 && 分别实现

```
$ cat > test.sh
#!/bin/bash
echo "test"
[CTRL+D]  # 按下组合键CTRL与D结束cat输入，后同，不再注明
$ chmod +x test.sh
$ if test -s test.sh -a -x test.sh; then echo "YES"; else echo "NO"; fi
YES
$ if test -s test.sh && test -x test.sh; then echo "YES"; else echo "NO"; fi
YES
```

范例：要求某个字符串要么为空，要么和某个字符串相等

```
$ str1="test"
$ str2="test"
$ if test -z "$str2" -o "$str2" == "$str1"; then echo "YES"; else echo "NO"; fi
YES
$ if test -z "$str2" || test "$str2" == "$str1"; then echo "YES"; else echo "NO"; fi
YES
```

范例：测试某个数字不满足指定的所有条件

```
$ i=5
$ if test ! $i -lt 5 -a $i -ne 6; then echo "YES"; else echo "NO"; fi
YES
$ if ! test $i -lt 5 -a $i -eq 6; then echo "YES"; else echo "NO"; fi
YES
```

很容易找出它们的区别， `-a` 和 `-o` 作为测试命令的参数用在测试命令的内部，而 `&&` 和 `||` 则用来运算测试的返回值， `!` 为两者通用。需要关注的是：

- 有时可以不用 `!` 运算符，比如 `-eq` 和 `-ne` 刚好相反，可用于测试两个数值是否相等； `-z` 与 `-n` 也是对应的，用来测试某个字符串是否为空
- 在 `Bash` 里， `test` 命令可以用[] 运算符取代，但是需要注意，[`之后与`] 之前需要加上额外的空格
- 在测试字符串时，所有变量建议用双引号包含起来，以防止变量内容为空时出现仅有测试参数，没有测试内容的情况

下面我们用实例来演示上面三个注意事项：

- `-ne` 和 `-eq` 对应的，我们有时候可以免去 `!` 运算

```
$ i=5
$ if test $i -eq 5; then echo "YES"; else echo "NO"; fi
YES
$ if test $i -ne 5; then echo "YES"; else echo "NO"; fi
NO
$ if test ! $i -eq 5; then echo "YES"; else echo "NO"; fi
NO
```

- 用 `[ ]` 可以取代 `test`，这样看上去会“美观”很多

```
$ if [ $i -eq 5 ]; then echo "YES"; else echo "NO"; fi
YES
$ if [ $i -gt 4 ] && [ $i -lt 6 ]; then echo "YES"; else echo "NO"; fi
YES
```

- 记得给一些字符串变量加上 `""`，记得 `[` 之后与 `]` 之前多加一个空格

```
$ str=""
$ if [ "$str" = "test"]; then echo "YES"; else echo "NO"; fi
-bash: [: missing `]'
NO
$ if [ $str = "test" ]; then echo "YES"; else echo "NO"; fi
-bash: [: =: unary operator expected
NO
$ if [ "$str" = "test" ]; then echo "YES"; else echo "NO"; fi
NO
```

到这里，条件测试就介绍完了，下面介绍命令列表，实际上在上面我们已经使用过了，即多个test命令的组合，通过 `&&`，`||` 和 `!` 组合起来的命令序列。这种命令序列可以有效替换 `if/then` 的条件分支结构。这不难想到我们在 C 语言程序设计中经常做的如下的选择题（很无聊的例子，但是有意义）：下面是否会打印 `j`，如果打印，将打印什么？

```
#include <stdio.h>
int main()
{
    int i, j;

    i=5;j=1;
    if ((i==5) && (j=5))  printf("%d\n", j);

    return 0;
}
```

很容易知道将打印数字 5，因为 `i==5` 这个条件成立，而且随后是 `&&`，要判断整个条件是否成立，我们得进行后面的判断，可是这个判断并非常规的判断，而是先把 `j` 修改为 5，再转换为真值，所以条件为真，打印出 5 。因此，这句可以解释为：如果 `i` 等于 5，那么把 `j` 赋值为 5，如果 `j` 大于 1 （因为之前已经为真），那么打印出 `j` 的值。这样用 `&&` 连结起来的判断语句替代了两个 `if` 条件分支语句。

正是基于逻辑运算特有的性质，我们可以通过 `&&`，`||` 来取代 `if/then` 等条件分支结构，这样就产生了命令列表。

命令列表

命令列表的执行规律

命令列表的执行规律符合逻辑运算的运算规律，用 `&&` 连接起来的命令，如果前者成功返回，将执行后面的命令，反之不然；用 `||` 连接起来的命令，如果前者成功返回，将不执行后续命令，反之不然。

范例：如果 **ping** 通 **www.lzu.edu.cn**，那么打印连通信息

```
$ ping -c 1 www.lzu.edu.cn -W 1 && echo "=======connected======="
```

非常有趣的问题出来了，即我们上面已经提到的：为什么要让 C 程序在 `main()` 函数的最后返回 0 ？如果不这样，把这种程序放入命令列表会有什么样的结果？你自己写个简单的 C 程序，然后放入命令列表看看。

命令列表的作用

有时用命令列表取代 `if/then` 等条件分支结构可以省掉一些代码，而且使得程序比较美观、易读，例如：

范例：在脚本里判断程序的参数个数，和参数类型

```
#!/bin/bash

echo $#
echo $1
if [ $# -eq 1 ] && (echo $1 | grep '^[0-9]*$' >/dev/null);then
    echo "YES"
fi
```

说明：上例要求参数个数为 1 并且类型为数字。

再加上 `exit 1`，我们将省掉 `if/then` 结构

```
#!/bin/bash

echo $#
echo $1
! ([ $# -eq 1 ] && (echo $1 | grep '^[0-9]*$' >/dev/null)) && exit 1

echo "YES"
```

这样处理后，对程序参数的判断仅仅需要简单的一行代码，而且变得更美观。

小结

这一节介绍了 Shell 编程中的逻辑运算，条件测试和命令列表。

字符串操作

前言

忙活了一个礼拜，终于等到周末，可以空下来写点东西。

之前已经完成《数值运算》和《布尔运算》，这次轮到介绍字符串操作 。咱们先得弄明白两个内容：

- 什么是字符串？
- 对字符串有哪些操作？

下面是"在线新华字典"的解释：

> 字符串: 简称“串”。有限字符的序列。数据元素为字符的线性表，是一种数据的逻辑结构。在计算机中可有不同的存储结构。在串上可进行求子串、插入字符、删除字符、置换字符等运算。

而字符呢？

> 字符: 计算机程序设计及操作时使用的符号。包括字母、数字、空格符、提示符及各种专用字符等。

照这样说，之前介绍的数值运算中的数字，布尔运算中的真假值，都是以字符的形式呈现出来的，是一种特别的字符，对它们的运算只不过是字符操作的特例罢了。而这里将研究一般字符的运算，它具有非常重要的意义，因为对我们来说，一般的工作都是处理字符而已。这些运算实际上将围绕上述两个定义来做，它们包括：

- 找出字符或者字符串的类型，是数字、字母还是其他特定字符，是可打印字符，还是不可打印字符（一些控制字符）。
- 找出组成字符串的字符个数和字符串的存储结构（比如数组）。
- 对串的常规操作：求子串、插入字符、删除字符、置换字符、字符串的比较等。
- 对串的一些比较复杂而有趣的操作，这里将在最后介绍一些有趣的范例。

字符串的属性

字符串的类型

字符有可能是数字、字母、空格、其他特殊字符，而字符串有可能是它们中的一种或者多种的组合，在组合之后还可能形成具有特定意义的字符串，诸如邮件地址，URL地址等。

范例：数字或者数字组合

```
$ i=5;j=9423483247234;
$ echo $i | grep -q "^[0-9]$"
$ echo $?
0
$ echo $j | grep -q "^[0-9]\+$"
$ echo $?
0
```

范例：字符组合（小写字母、大写字母、两者的组合）

```
$ c="A"; d="fwefewjuew"; e="fewfEFWefwefe"
$ echo $c | grep -q "^[A-Z]$"
$ echo $d | grep -q "^[a-z]\+$"
$ echo $e | grep -q "^[a-zA-Z]\+$"
```

范例：字母和数字的组合

```
$ ic="432fwfwefeFWEwefwef"
$ echo $ic | grep -q "^[0-9a-zA-Z]\+$"
```

范例：空格或者 **Tab** 键等

```
$ echo " " | grep " "
$ echo -e "\t" | grep "[[:space:]]" #[[:space:]]会同时匹配空格和TAB键
$ echo -e " \t" | grep "[[:space:]]"
$ echo -e "\t" | grep "" #为在键盘上按下TAB键，而不是字符
```

范例：匹配邮件地址

```
$ echo "test2007@lzu.cn" | grep "[0-9a-zA-Z\.]*@[0-9a-zA-Z\.]*"
test2007@lzu.cn
```

范例：匹配 URL 地址(以 http 链接为例）

```
$ echo "http://news.lzu.edu.cn/article.jsp?newsid=10135" | grep "^http://[0-9a-zA-Z\./
=?]\+$"
http://news.lzu.edu.cn/article.jsp?newsid=10135
```

说明：

- `/dev/null` 和 `/dev/zero` 设备非常有趣，都犹如黑洞，什么东西掉进去都会消失殆尽；后者还是个能源箱，总能从那里取到0，直到退出
- `[[:space:]]` 是 `grep` 用于匹配空格或 TAB 键字符的标记，其他标记请查帮助：`man grep`
- 上面都是用 `grep` 来进行模式匹配，实际上 `sed`，`awk` 都可用来做模式匹配，关于匹配中用到的正则表达式知识，请参考后面的相关资料
- 如果想判断字符串是否为空，可判断其长度是否为零，可通过 `test` 命令的 `-z` 选项来实现，具体用法见 `test` 命令，`man test`

范例：判断字符是否为可打印字符

```
$ echo "\t\n" | grep "[[:print:]]"
\t\n
$ echo $?
0
$ echo -e "\t\n" | grep "[[:print:]]"
$ echo $?
1
```

字符串的长度

除了组成字符串的字符类型外，字符串还有哪些属性呢？组成字符串的字符个数。

下面我们来计算字符串的长度，即所有字符的个数，并简单介绍几种求字符串中指定字符个数的方法。

范例：计算某个字符串的长度

即计算所有字符的个数，计算方法五花八门，择其优着而用之：

```
$ var="get the length of me"
$ echo ${var}     # 这里等同于$var
get the length of me
$ echo ${#var}
20
$ expr length "$var"
20
$ echo $var | awk '{printf("%d\n", length($0));}'
20
$ echo -n $var |  wc -c
20
```

范例：计算某些指定字符或者字符组合的个数

```
$ echo $var | tr -cd g | wc -c
2
$ echo -n $var | sed -e 's/[^g]//g' | wc -c
2
$ echo -n $var | sed -e 's/[^gt]//g' | wc -c
5
```

范例：统计单词个数

更多相关信息见《数值计算》的单词统计 相关范例。

```
$ echo $var | wc -w
5
$ echo "$var" | tr " " "\n" | grep get | uniq -c
1
$ echo "$var" | tr " " "\n" | grep get | wc -l
1
```

说明：

`${}` 操作符在 Bash 里头是一个“大牛”，能胜任相当多的工作，具体就看网中人的《Shell十三问》之 `$(())` 与 `$()` 还有 `${}` 差在哪？"吧。

字符串的显示

接下来讨论如何控制字符在终端的显示。

范例：在屏幕控制字符显示位置、颜色、背景等

```
$ echo -e "\033[31;40m" #设置前景色为黑色，背景色为红色
$ echo -e '\033[11;29H Hello, World!' #在屏幕的第11行，29列开始打印字符串Hello,World!
```

范例：在屏幕的某个位置动态显示当前系统时间

```
$ while :; do echo -e "\033[11;29H "$(date "+%Y-%m-%d %H:%M:%S"); done
```

范例：过滤掉某些控制字符串

用 `col` 命令过滤某些控制字符，在处理诸如 `script`，`screen` 等截屏命令的输出结果时，很有用。

```
$ screen -L
$ cat /bin/cat
$ exit
$ cat screenlog.0 | col -b   # 把一些控制字符过滤后，就可以保留可读的操作日志
```

字符串的存储

在我们看来，字符串是一连串的字符而已，但是为了操作方便，我们往往可以让字符串呈现出一定的结构。在这里，我们不关心字符串在内存中的实际存储结构，仅仅关系它呈现出来的逻辑结构。比如，这样一个字符串： `"get the length of me"`，我们可以从不同的方面来呈现它。

- 通过字符在串中的位置来呈现它

这样我们就可以通过指定位置来找到某个子串。这在 C 语言中通常可以利用指针来做。而在 Shell 编程中，有很多可用的工具，诸如 `expr`，`awk` 都提供了类似方法来实现子串的查询动作。两者都几乎支持模式匹配 `match` 和完全匹配 `index`。这在后面的字符串操作中将详细介绍。

- 根据某个分割符来取得字符串的各个部分

这里最常见的就是行分割符、空格或者 `TAB` 分割符了，前者用来当行号，我们似乎已经司空见惯了，因为我们的编辑器就这样“莫名”地处理着行分割符（在 UNIX 下为 `\n`，在其他系统下有一些不同，比如 Windows 下为 `\r\n`）。而空格或者 `TAB` 键经常用来分割数据库的各个字段，这似乎也是司空见惯的事情。

正因为这样，所以产生了大量优秀的行编辑工具，诸如 `grep`，`awk`，`sed` 等。在“行内”（姑且这么说吧，就是处理单行，即字符串中不再包含行分割符）的字符串分割方面，`cut` 和 `awk` 提供了非常优越的“行内”（处理单行）处理能力。

- 更方便地处理用分割符分割好的各个部分

同样是用到分割符，但为了更方便的操作分割以后的字符串的各个部分，我们抽象了“数组”这么一个数据结构，从而让我们更加方便地通过下标来获取某个指定的部分。 `bash` 提供了这么一种数据结构，而优秀的 `awk` 也同样提供了它，我们这里将简单介绍它们的用法。

范例：把字符串拆分成字符串数组

- Bash 提供的数组数据结构，以数字为下标的，和 C 语言从 0 开始的下标一样

  ```
  $ var="get the length of me"
  $ var_arr=($var)    #把字符串var存放到字符串数组var_arr中，默认以空格作为分割符
  $ echo ${var_arr[0]} ${var_arr[1]} ${var_arr[2]} ${var_arr[3]} ${var_arr[4]}
  get the length of me
  $ echo ${var_arr[@]}    #整个字符串，可以用*代替@，下同
  get the length of me
  $ echo ${#var_arr[@]}   #类似于求字符串长度，`#`操作符也可用来求数组元素个数
  5
  ```

  也可以直接给某个数组元素赋值

  ```
  $ var_arr[5]="new_element"
  $ echo ${#var_arr[@]}
  6
  $ echo ${var_arr[5]}
  new_element
  ```

  Bash 实际上还提供了一种类似于“数组”的功能，即 `for i in` ，它可以很方便地获取某个字符串的各个部分，例如：

  ```
  $ for i in $var; do echo -n $i"_"; done
  get_the_length_of_me_
  ```

- `awk` 里的数组，注意比较它和 `Bash` 里的数组的异同

  `split` 把一行按照空格分割，存放到数组 `var_arr` 中，并返回数组长度。注意：这里第一个元素下标不是 0，而是 1

  ```
  $ echo $var | awk '{printf("%d %s\n", split($0, var_arr, " "), var_arr[1]);}'
  5 get
  ```

  实际上，上述操作很类似 `awk` 自身的行处理功能： `awk` 默认把一行按照空格分割为多个域，并可以通过 `$1` ， `$2` ， `$3` `...` 来获取， `$0` 表示整行。

> 这里的 `NF` 是该行的域的总数，类似于上面数组的长度，它同样提供了一种通过类似“下标”访问某个字符串的功能。
>
> ```
> $ echo $var | awk '{printf("%d | %s %s %s %s %s | %s\n", NF, $1, $2, $3, $4, $5, $
> 0);}'
> 5 | get the length of me | get the length of me
> ```
>
> `awk` 的“数组”功能何止于此呢，看看它的 `for` 引用吧，注意，这个和 `Bash` 里头的 `for` 不太一样， `i` 不是元素本身，而是下标：
>
> ```
> $ echo $var | awk '{split($0, var_arr, " "); for(i in var_arr) printf("%s ",var_ar
> r[i]);}'
> of me get the length
> 4 5 1 2 3
> ```
>
> 另外，从上述结果可以看到，经过 `for` 处理后，整个结果没有按照原理的字符顺序排列，不过如果仅仅是迭代出所有元素这个同样很有意义。

`awk` 还有更“厉害”的处理能力，它的下标可以不是数字，可以是字符串，从而变成了“关联”数组，这种“关联”在某些方面非常方便。 比如，把某个文件中的某个系统调用名根据另外一个文件中的函数地址映射表替换成地址，可以这么实现：

```
$ cat symbol
sys_exit
sys_read
sys_close
$ ls /boot/System.map*
/boot/System.map-2.6.20-16-generic
$ awk '{if(FILENAME ~ "System.map") map[$3]=$1; else {printf("%s\n", map[$1])}}' \
    /boot/System.map-2.6.20-16-generic symbol
c0129a80
c0177310
c0175d80
```

另外，awk还支持用delete函数删除某个数组元素。如果某些场合有需要的话，别忘了awk还支持二维数组。

字符串常规操作

字符串操作包括取子串、查询子串、插入子串、删除子串、子串替换、子串比较、子串排序、子串进制转换、子串编码转换等。

取子串

取子串的方法主要有：

- 直接到指定位置求子串
- 字符匹配求子串

范例：按照位置取子串

比如从什么位置开始，取多少个字符

```
$ var="get the length of me"
$ echo ${var:0:3}
get
$ echo ${var:(-2)}   # 方向相反呢
me

$ echo `expr substr "$var" 5 3` #记得把$var引起来，否则expr会因为空格而解析错误
the

$ echo $var | awk '{printf("%s\n", substr($0, 9, 6))}'
length
```

`awk` 把 `$var` 按照空格分开为多个变量，依次为 `$1`，`$2`，`$3`，`$4`，`$5`

```
$ echo $var | awk '{printf("%s\n", $1);}'
get
$ echo $var | awk '{printf("%s\n", $5);}'
me
```

差点略掉 `cut` 小工具，它用起来和awk类似，`-d` 指定分割符，如同 `awk` 用 `-F` 指定分割符一样；`-f` 指定“域”，如同awk的 `$数字` 。

```
$ echo $var | cut -d" " -f 5
```

范例：匹配字符求子串

用 Bash 内置支持求子串：

```
$ echo ${var%% *} #从右边开始计算，删除最左边的空格右边的所有字符
get
$ echo ${var% *} #从右边开始计算，删除第一个空格右边的所有字符
get the length of
$ echo ${var##* }  #从左边开始计算，删除最右边的空格左边的所有字符
me
$ echo ${var#* }  #从左边开始计算，删除第一个空格左边的所有字符
the length of me
```

删除所有 `空格+字母组合` 的字符串：

```
$ echo $var | sed 's/ [a-z]*//g'
get
$ echo $var | sed 's/[a-z]* //g'
me
```

`sed` 有按地址（行）打印(p)的功能，记得先用 `tr` 把空格换成行号：

```
$ echo $var | tr " " "\n" | sed -n 1p
get
$ echo $var | tr " " "\n" | sed -n 5p
me
```

`tr` 也可以用来取子串，它可以类似 `#` 和 `%` 来“拿掉”一些字符串来实现取子串：

```
$ echo $var | tr -d " "
getthelengthofme
$ echo $var | tr -cd "[a-z]" #把所有的空格都拿掉了，仅仅保留字母字符串，注意-c和-d的用法
getthelengthofme
```

说明：

- `%` 和 `#` 删除字符的方向不一样，前者在右，后者在左， `%%` 和 `%` ， `##` 和 `#` 的方向是前者是最大匹配，后者是最小匹配。（好的记忆方法见网中人的键盘记忆法： `#` ， `$` ， `%` 是键盘依次从左到右的三个键）
- `tr` 的 `-c` 选项是 `complement` 的缩写，即 `invert` ，而 `-d` 选项是删除， `tr -cd "[a-z]"` 这样一来就变成保留所有的字母

对于字符串的截取，实际上还有一些命令，如果 `head` ， `tail` 等可以实现有意思的功能，可以截取某个字符串的前面、后面指定的行数或者字节数。例如：

```
$ echo "abcdefghijk" | head -c 4
abcd
$ echo -n "abcdefghijk" | tail -c 4
hijk
```

查询子串

子串查询包括：

- 返回符合某个模式的子串本身
- 返回子串在目标串中的位置

准备：在进行下面的操作之前，请准备一个文件 test.txt，里头有内容 "consists of"，用于下面的范例。

范例：查询子串在目标串中的位置

`expr index`貌似仅仅可以返回某个字符或者多个字符中第一个字符出现的位置

```
$ var="get the length of me"
$ expr index "$var" t
3
```

awk却能找出字串，match还可以匹配正则表达式

```
$ echo $var | awk '{printf("%d\n", match($0,"the"));}'
5
```

范例：查询子串，返回包含子串的行

`awk`，`sed` 都可以实现这些功能，但是 `grep` 最擅长

```
$ grep "consists of" test.txt   # 查询文件包含consists of的行，并打印这些行
$ grep "consists[[:space:]]of" -n -H test.txt # 打印文件名，子串所在行的行号和该行的内容
$ grep "consists[[:space:]]of" -n -o test.txt # 仅仅打印行号和匹配到的子串本身的内容

$ awk '/consists of/{ printf("%s:%d:%s\n",FILENAME, FNR, $0)}' text  #看到没？和grep的结果一样
$ sed -n -e '/consists of/=;/consists of/p' text #同样可以打印行号
```

说明：

- `awk`，`grep`，`sed` 都能通过模式匹配查找指定字符串，但它们各有所长，将在后续章节中继续使用和比较它们，进而发现各自优点
- 在这里姑且把文件内容当成了一个大的字符串，在后面章节中将专门介绍文件操作，所以对文件内容中存放字符串的操作将会有更深入的分析和介绍

子串替换

子串替换就是把某个指定的子串替换成其他的字符串，这里蕴含了“插入子串”和“删除子串”的操作。例如，想插入某个字符串到某个子串之前，就可以把原来的子串替换成”子串+新的字符串“，如果想删除某个子串，就把子串替换成空串。不过有些工具提供了一些专门的用法来做

插入子串和删除子串的操作，所以呆伙还会专门介绍。另外，要想替换掉某个子串，一般都是先找到子串（查询子串），然后再把它替换掉，实质上很多工具在使用和设计上都体现了这么一点。

范例：把变量 var 中的空格替换成下划线

用 `{}` 运算符，还记得么？网中人的教程

```
$ var="get the length of me"
$ echo ${var/ /_}        #把第一个空格替换成下划线
get_the length of me
$ echo ${var// /_}       #把所有空格都替换成下划线
get_the_length_of_me
```

用 `awk` ，`awk` 提供了转换的最小替换函数 `sub` 和全局替换函数 `gsub` ，类似 `/` 和 `//`

```
$ echo $var | awk '{sub(" ", "_", $0); printf("%s\n", $0);}'
get_the length of me
$ echo $var | awk '{gsub(" ", "_", $0); printf("%s\n", $0);}'
get_the_length_of_me
```

用 `sed` ，子串替换可是 `sed` 的特长：

```
$ echo $var | sed -e 's/ /_/'    #s <= substitude
get_the length of me
$ echo $var | sed -e 's/ /_/g'   #看到没有，简短两个命令就实现了最小匹配和最大匹配g <= global
get_the_length_of_me
```

有忘记 `tr` 命令么？可以用替换单个字符的：

```
$ echo $var | tr " " "_"
get_the_length_of_me
$ echo $var | tr '[a-z]' '[A-Z]'   #这个可有意思了，把所有小写字母都替换为大写字母
GET THE LENGTH OF ME
```

说明： `sed` 还有很有趣的标签用法呢，下面再介绍吧。

有一种比较有意思的字符串替换是：整个文件行的倒置，这个可以通过 `tac` 命令实现，它会把文件中所有的行全部倒转过来。在某种意义上来说，排序实际上也是一个字符串替换。

插入子串

在指定位置插入子串，这个位置可能是某个子串的位置，也可能是从某个文件开头算起的某个长度。通过上面的练习，我们发现这两者之间实际上是类似的。

公式：插入子串=把"old子串"替换成"old子串+new子串"或者"new子串+old子串"

范例：在 var 字符串的空格之前或之后插入一个下划线

用{}：

```
$ var="get the length of me"
$ echo ${var/ /_ }        #在指定字符串之前插入一个字符串
get_ the length of me
$ echo ${var// /_ }
get_ the_ length_ of_ me
$ echo ${var/ / _}        #在指定字符串之后插入一个字符串
get _the length of me
$ echo ${var// / _}
get _the _length _of _me
```

其他的还用演示么？这里主要介绍sed怎么用来插入字符吧，因为它的标签功能很有趣 说明：`(` 和 `)` 将匹配到的字符串存放为一个标签，按匹配顺序为 `\1`，`\2`...

```
$ echo $var | sed -e 's/\( \)/_\1/'
get_ the length of me
$ echo $var | sed -e 's/\( \)/_\1/g'
get_ the_ length_ of_ me
$ echo $var | sed -e 's/\( \)/\1_/'
get _the length of me
$ echo $var | sed -e 's/\( \)/\1_/g'
get _the _length _of _me
```

看看 `sed` 的标签的顺序是不是 `\1`，`\2`...，看到没？`\2` 和 `\1` 调换位置后，`the` 和 `get` 的位置掉换了：

```
$ echo $var | sed -e 's/\([a-z]*\) \([a-z]*\) /\2 \1 /g'
the get of length me
```

`sed` 还有专门的插入指令，`a` 和 `i`，分别表示在匹配的行后和行前插入指定字符

```
$ echo $var | sed '/get/a test'
get the length of me
test
$ echo $var | sed '/get/i test'
test
get the length of me
```

删除子串

删除子串：应该很简单了吧，把子串替换成“空”（什么都没有）不就变成了删除么。还是来简单复习一下替换吧。

范例：把 var 字符串中所有的空格给删除掉。

鼓励：这样一替换不知道变成什么单词啦，谁认得呢？但是中文却是连在一起的，所以中文有多难，你想到了么？原来你也是个语言天才，而英语并不可怕，你有学会它的天赋，只要有这个打算。

再用 `{}`

```
$ echo ${var// /}
getthelengthofme
```

再用 `awk`

```
$ echo $var | awk '{gsub(" ","",$0); printf("%s\n", $0);}'
```

再用 `sed`

```
$ echo $var | sed 's/ //g'
getthelengthofme
```

还有更简单的 `tr` 命令，`tr` 也可以把空格给删除掉，看

```
$ echo $var | tr -d " "
getthelengthofme
```

如果要删除第一个空格后面所有的字符串该怎么办呢？还记得 `{}` 的 `#` 和 `%` 用法么？如果不记得，回到这节的开头开始复习吧。（实际上删除子串和取子串未尝不是两种互补的运算呢，删除掉某些不想要的子串，也就同时取得另外那些想要的子串——这个世界就是一个“二元”的世界，非常有趣）

子串比较

这个很简单：还记得 `test` 命令的用法么？ `man test` 。它可以用来判断两个字符串是否相等。另外，有发现“字符串是否相等”和“字符串能否跟另外一个字符串匹配 `"` 两个问题之间的关系吗？如果两个字符串完全匹配，那么这两个字符串就相等了。所以呢，上面用到的字符串匹配方法，也同样可以用到这里。

子串排序

差点忘记这个重要内容了，子串排序可是经常用到，常见的有按字母序、数字序等正序或反序排列。 `sort` 命令可以用来做这个工作，它和其他行处理命令一样，是按行操作的，另外，它类似 `cut` 和 `awk` ，可以指定分割符，并指定需要排序的列。

```
$ var="get the length of me"
$ echo $var | tr ' ' '\n' | sort   #正序排
get
length
me
of
the
$ echo $var | tr ' ' '\n' | sort -r #反序排
the
of
me
length
get
$ cat > data.txt
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
41 45 44 44 26 44 42 20 20 38 37 25 45 45 45
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
44 20 30 39 35 38 38 28 25 30 36 20 24 32 33
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
41 33 51 39 20 20 44 37 38 39 42 40 37 50 50
46 47 48 49 50 51 52 53 54 55 56
42 43 41 42 45 42 19 39 75 17 17
$ cat data.txt | sort -k 2 -n
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
44 20 30 39 35 38 38 28 25 30 36 20 24 32 33
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
41 33 51 39 20 20 44 37 38 39 42 40 37 50 50
42 43 41 42 45 42 19 39 75 17 17
41 45 44 44 26 44 42 20 20 38 37 25 45 45 45
46 47 48 49 50 51 52 53 54 55 56
```

子串进制转换

如果字母和数字字符用来计数，那么就存在进制转换的问题。在《数值计算》一节，已经介绍了 `bc` 命令，这里再复习一下。

```
$ echo "ibase=10;obase=16;10" | bc
A
```

说明： `ibase` 指定输入进制， `obase` 指出输出进制，这样通过调整 `ibase` 和 `obase` ，你想怎么转就怎么转啦！

子串编码转换

什么是字符编码？这个就不用介绍了吧，看过那些乱七八糟显示的网页么？大多是因为浏览器显示时的”编码“和网页实际采用的”编码“不一致导致的。字符编码通常是指：把一序列”可打印“字符转换成二进制表示，而字符解码呢则是执行相反的过程，如果这两个过程不匹配，则出现了所谓的”乱码“。

为了解决”乱码“问题呢？就需要进行编码转换。在 Linux 下，我们可以使用 `iconv` 这个工具来进行相关操作。这样的情况经常在不同的操作系统之间移动文件，不同的编辑器之间交换文件的时候遇到，目前在 Windows 下常用的汉字编码是 `gb2312` ，而在 Linux 下则大多采用 `utf8` 。

```
$ nihao_utf8=$(echo "你好")
$ nihao_gb2312=$(echo $nihao_utf8 | iconv -f utf8 -t gb2312)
```

字符串操作进阶

实际上，在用 Bash 编程时，大部分时间都是在处理字符串，因此把这一节熟练掌握非常重要。

正则表达式

范例：处理 URL 地址

URL 地址(URL（Uniform Resoure Locator：统一资源定位器）是WWW页的地址)几乎是我们日常生活的玩伴，我们已经到了无法离开它的地步啦，对它的操作很多，包括判断 URL 地址的有效性，截取地址的各个部分（服务器类型、服务器地址、端口、路径等）并对各个部分进行进一步的操作。

下面我们来具体处理这个URL地址：ftp://anonymous:ftp@mirror.lzu.edu.cn/software/scim-1.4.7.tar.gz

```
$ url="ftp://anonymous:ftp@mirror.lzu.edu.cn/software/scim-1.4.7.tar.gz"
```

匹配URL地址，判断URL地址的有效性

```
$ echo $url | grep "ftp://[a-z]*:[a-z]*@[a-z\./-]*"
```

截取服务器类型

```
$ echo ${url%%:*}
ftp
$ echo $url | cut -d":" -f 1
ftp
```

截取域名

```
$ tmp=${url##*@} ; echo ${tmp%%/*}
mirror.lzu.edu.cn
```

截取路径

```
$ tmp=${url##*@} ; echo ${tmp%/*}
mirror.lzu.edu.cn/software
```

截取文件名

```
$ basename $url
scim-1.4.7.tar.gz
$ echo ${url##*/}
scim-1.4.7.tar.gz
```

截取文件类型（扩展名）

```
$ echo $url | sed -e 's/.*[0-9].\(.*\)/\1/g'
tar.gz
```

范例：匹配某个文件中的特定范围的行

先准备一个测试文件README

```
Chapter 7 -- Exercises

7.1 please execute the program: mainwithoutreturn, and print the return value
of it with the command "echo $?", and then compare the return of the printf
function, they are the same.

7.2 it will depend on the exection mode, interactive or redirection to a file,
if interactive, the "output" action will accur after the \n char with the line
buffer mode, else, it will be really "printed" after all of the strings have
been stayed in the buffer.

7.3 there is no another effective method in most OS. because argc and argv are
not global variables like environ.
```

然后开始实验，

打印出答案前指定行范围：第 7 行到第 9 行，刚好找出了第 2 题的答案

```
$ sed -n 7,9p README
7.2 it will depend on the exection mode, interactive or redirection to a file,
if interactive, the "output" action will accur after the \n char with the line
buffer mode, else, it will be really "printed" after all of the strings have
```

其实，因为这个文件内容格式很有特色，有更简单的办法

```
$ awk '/7.2/,/^$/ {printf("%s\n", $0);}' README
7.2 it will depend on the exection mode, interactive or redirection to a file,
if interactive, the "output" action will accur after the \n char with the line
buffer mode, else, it will be really "printed" after all of the strings have
been stayed in the buffer.
```

有了上面的知识，就可以非常容易地进行这些工作啦：修改某个文件的文件名，比如调整它的编码，下载某个网页里头的所有 `pdf` 文档等。这些就作为练习自己做吧。

处理格式化的文本

平时做工作，大多数时候处理的都是一些“格式化”的文本，比如类似 `/etc/passwd` 这样的有固定行和列的文本，也有类似 `tree` 命令输出的那种具有树形结构的文本，当然还有其他具有特定结构的文本。

关于树状结构的文本的处理，可以参考我早期写的另外一篇博客文章：源码分析：静态分析 C 程序函数调用关系图

实际上，只要把握好特性结构的一些特点，并根据具体的应用场合，处理起来就不会困难。

下面来介绍具体文本的操作，以 `/etc/passwd` 文件为例。关于这个文件的帮忙和用法，请通过 `man 5 passwd` 查看。下面对这个文件以及相关的文件进行一些有意义的操作。

范例：选取指定列

选取/etc/passwd文件中的用户名和组ID两列

```
$ cat /etc/passwd | cut -d":" -f1,4
```

选取/etc/group文件中的组名和组ID两列

```
$ cat /etc/group | cut -d":" -f1,3
```

范例：文件关联操作

如果想找出所有用户所在的组，怎么办？

```
$ join -o 1.1,2.1 -t":" -1 4 -2 3 /etc/passwd /etc/group
root:root
bin:bin
daemon:daemon
adm:adm
lp:lp
pop:pop
nobody:nogroup
falcon:users
```

说明： `join` 命令用来连接两个文件，有点类似于数据库的两个表的连接。 `-t` 指定分割符， `-1 4 -2 3` 指定按照第一个文件的第 4 列和第二个文件的第 3 列，即组 `ID` 进行连接， `-o` `1.1,2.1` 表示仅仅输出第一个文件的第一列和第二个文件的第一列，这样就得到了我们要的结果，不过，可惜的是，这个结果并不准确，再进行下面的操作，你就会发现：

```
$ cat /etc/passwd | sort -t":" -n -k 4 > /tmp/passwd
$ cat /etc/group | sort -t":" -n -k 3 > /tmp/group
$ join -o 1.1,2.1 -t":" -1 4 -2 3 /tmp/passwd /tmp/group
halt:root
operator:root
root:root
shutdown:root
sync:root
bin:bin
daemon:daemon
adm:adm
lp:lp
pop:pop
nobody:nogroup
falcon:users
games:users
```

可以看到这个结果才是正确的，所以以后使用 `join` 千万要注意这个问题，否则采取更保守的做法似乎更能保证正确性，更多关于文件连接的讨论见参考后续资料。

上面涉及到了处理某格式化行中的指定列，包括截取（如 `SQL` 的 `select` 用法），连接（如 `SQL` 的 `join` 用法），排序（如 `SQL` 的 `order by` 用法），都可以通过指定分割符来拆分某个格式化的行，另外，“截取”的做法还有很多，不光是 `cut` ， `awk` ，甚至通过 `IFS` 指定分割符的 `read` 命令也可以做到，例如：

```
$ IFS=":"; cat /etc/group | while read C1 C2 C3 C4; do echo $C1 $C3; done
```

因此，熟悉这些用法，我们的工作将变得非常灵活有趣。

到这里，需要做一个简单的练习，如何把按照列对应的用户名和用户 ID 转换成按照行对应的，即把类似下面的数据：

```
$ cat /etc/passwd | cut -d":" -f1,3 --output-delimiter=" "
root 0
bin 1
daemon 2
```

转换成：

```
$ cat a
root    bin     daemon
0       1       2
```

并转换回去，有什么办法呢？记得诸如 `tr` ， `paste` ， `split` 等命令都可以使用。

参考方法：

- 正转换：先截取用户名一列存入文件 `user`，再截取用户 `ID` 存入 `id`，再把两个文件用 `paste -s` 命令连在一起，这样就完成了正转换
- 逆转换：先把正转换得到的结果用 `split -1` 拆分成两个文件，再把两个拆分后的文件用 `tr` 把分割符 `\t` 替换成 `\n`，只有用 `paste` 命令把两个文件连在一起，这样就完成了逆转换。

参考资料

- 《高级 Bash 脚本编程指南》之操作字符串，之指定变量的类型
- 《Shell十三问》之 `$(())` 与 `$()` 还有 `${}` 差在哪？
- Regular Expressions - User guide
- Regular Expression Tutorial
- Grep Tutorial
- Sed Tutorial
- awk Tutorial
- sed Tutorial
- An awk Primer
- 一些奇怪的 UNIX 指令名字的由来
- 磨练构建正则表达式模式的技能
- 基础11：文件分类、合并和分割(sort,uniq,join,cut,paste,split)
- 使用 Linux 文本工具简化数据的提取
- SED 单行脚本快速参考（Unix流编辑器）

后记

- 这一节本来是上个礼拜该弄好的，但是这些天太忙了，到现在才写好一个“初稿”，等到有时间再补充具体的范例。这一节的范例应该是最最有趣的，所有得好好研究一下几个有趣的范例。
- 写完上面的部分貌似是 1 点多，刚 `check` 了一下错别字和语法什么的，再添加了一节，即“字符串的存储结构”，到现在已经快 `half past 2` 啦，晚安，朋友们。
- 26 号，添加“子串进制转换”和“子串编码转换”两小节以及一个处理 `URL` 地址的范例。

文件操作

前言

这周来探讨文件操作。

在日常学习和工作中，总是在不断地和各种文件打交道，这些文件包括普通文本文件，可以执行的程序，带有控制字符的文档、存放各种文件的目录、网络套接字文件、设备文件等。这些文件又具有诸如属主、大小、创建和修改日期等各种属性。文件对应文件系统的一些数据块，对应磁盘等存储设备的一片连续空间，对应于显示设备却是一些具有不同形状的字符集。

在这一节，为了把关注点定位在文件本身，不会深入探讨文件系统以及存储设备是如何组织文件的（在后续章节再深入探讨），而是探讨对它最熟悉的一面，即把文件当成是一系列的字符（一个 `byte` ）集合看待。因此之前介绍的《Shell 编程范例之字符串操作》在这里将会得到广泛的应用，关于普通文件的读写操作已经非常熟练，那就是“重定向”，这里会把这部分独立出来介绍。关于文件在 Linux 下的“数字化”（文件描述符）高度抽象，“一切皆为文件”的哲学在 Shell 编程里也得到了深刻的体现。

下面先来介绍文件的各种属性，然后介绍普通文件的一般操作。

文件的各种属性

首先通过文件的结构体来看看文件到底有哪些属性：

```
struct stat {
    dev_t st_dev; /* 设备   */
    ino_t st_ino; /* 节点   */
    mode_t st_mode; /* 模式   */
    nlink_t st_nlink; /* 硬连接 */
    uid_t st_uid; /* 用户ID */
    gid_t st_gid; /* 组ID   */
    dev_t st_rdev; /* 设备类型 */
    off_t st_off; /* 文件字节数 */
    unsigned long  st_blksize; /* 块大小 */
    unsigned long st_blocks; /* 块数   */
    time_t st_atime; /* 最后一次访问时间 */
    time_t st_mtime; /* 最后一次修改时间 */
    time_t st_ctime; /* 最后一次改变时间(指属性) */
};
```

下面逐次来了解这些属性，如果需要查看某个文件属性，用 `stat` 命令就可，它会按照上面的结构体把信息列出来。另外， `ls` 命令在跟上一定参数后也可以显示文件的相关属性，比如 `-l` 参数。

文件类型

文件类型对应于上面的 `st_mode` , 文件类型有很多，比如常规文件、符号链接（硬链接、软链接）、管道文件、设备文件(符号设备、块设备)、socket文件等，不同的文件类型对应不同的功能和作用。

范例：在命令行简单地区分各类文件

```
$ ls -l
total 12
drwxr-xr-x 2 root root 4096 2007-12-07 20:08 directory_file
prw-r--r-- 1 root root    0 2007-12-07 20:18 fifo_pipe
brw-r--r-- 1 root root 3, 1 2007-12-07 21:44 hda1_block_dev_file
crw-r--r-- 1 root root 1, 3 2007-12-07 21:43 null_char_dev_file
-rw-r--r-- 2 root root  506 2007-12-07 21:55 regular_file
-rw-r--r-- 2 root root  506 2007-12-07 21:55 regular_file_hard_link
lrwxrwxrwx 1 root root   12 2007-12-07 20:15 regular_file_soft_link -> regular_file
$ stat directory_file/
  File: `directory_file/'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 301h/769d       Inode: 521521      Links: 2
Access: (0755/drwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2007-12-07 20:08:18.000000000 +0800
Modify: 2007-12-07 20:08:18.000000000 +0800
Change: 2007-12-07 20:08:18.000000000 +0800
$ stat null_char_dev_file
  File: `null_char_dev_file'
  Size: 0               Blocks: 0          IO Block: 4096   character special file
Device: 301h/769d       Inode: 521240      Links: 1     Device type: 1,3
Access: (0644/crw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2007-12-07 21:43:38.000000000 +0800
Modify: 2007-12-07 21:43:38.000000000 +0800
Change: 2007-12-07 21:43:38.000000000 +0800
```

说明：通过 `ls` 命令结果每行的第一个字符可以看到，它们之间都不相同，这正好反应了不同文件的类型。 `d` 表示目录， `-` 表示普通文件（或者硬链接）， `l` 表示符号链接， `p` 表示管道文件， `b` 和 `c` 分别表示块设备和字符设备（另外 `s` 表示 `socket` 文件）。在 `stat` 命令的结果中，可以在第二行的最后找到说明，从上面的操作可以看出， `directory_file` 是目录， `stat` 命令的结果中用 `directory` 表示，而 `null_char_dev_file` 它则用 `character special file` 说明。

范例：简单比较它们的异同

通常只会用到目录、普通文件、以及符号链接，很少碰到其他类型的文件，不过这些文件还是各有用处的，如果要做嵌入式开发或者进程通信等，可能会涉及到设备文件、有名管道（FIFO）。下面通过简单的操作来反应它们之间的区别（具体原理会在下一节《Shell 编程范例之文件系统》介绍，如果感兴趣，也可以提前到网上找找设备文件的作用、块设备和字符设备的区别、以及驱动程序中如何编写相关设备驱动等）。

对于普通文件：就是一系列字符的集合，所以可以读、写等

```
$ echo "hello, world" > regular_file
$ cat regular_file
hello, world
```

在目录中可以创建新文件，所以目录还有叫法：文件夹，到后面会分析目录文件的结构体，它实际上存放了它下面的各个文件的文件名。

```
$ cd directory_file
$ touch file1 file2 file3
```

对于有名管道，操作起来比较有意思：如果要读它，除非有内容，否则阻塞；如果要写它，除非有人来读，否则阻塞。它常用于进程通信中。可以打开两个终端 `terminal1` 和 `terminal2` ，试试看：

```
terminal1$ cat fifo_pipe #刚开始阻塞在这里，直到下面的写动作发生，才打印test字符串
terminal2$ echo "test" > fifo_pipe
```

关于块设备，字符设备，设备文件对应于 `/dev/hda1` 和 `/dev/null` ，如果用过 U 盘，或者是写过简单的脚本的话，这样的用法应该用过： :-)

```
$ mount hda1_block_dev_file /mnt #挂载硬盘的第一个分区到/mnt下（关于挂载的原理，在下一节讨论）
$ echo "fewfewfef" > /dev/null   #/dev/null像个黑洞，什么东西丢进去都消失殆尽
```

最后两个文件分别是 `regular_file` 文件的硬链接和软链接，去读写它们，他们的内容是相同的，不过去删除它们，他们却互不相干，硬链接和软链接又有何不同呢？前者可以说就是原文件，后者呢只是有那么一个 `inode` ，但没有实际的存储空间，建议用 `stat` 命令查看它们之间的区别，包括它们的 `Blocks` ， `inode` 等值，也可以考虑用 `diff` 比较它们的大小。

```
$ ls regular_file*
ls regular_file* -l
-rw-r--r-- 2 root root 204800 2007-12-07 22:30 regular_file
-rw-r--r-- 2 root root 204800 2007-12-07 22:30 regular_file_hard_link
lrwxrwxrwx 1 root root     12 2007-12-07 20:15 regular_file_soft_link -> regular_file
$ rm regular_file      # 删除原文件
$ cat regular_file_hard_link   # 硬链接还在，而且里头的内容还有呢
fefe
$ cat regular_file_soft_link
cat: regular_file_soft_link: No such file or directory
```

虽然软链接文件本身还在，不过因为它本身不存储内容，所以读不到东西，这就是软链接和硬链接的区别。

需要注意的是，硬链接不可以跨文件系统，而软链接则可以。另外，也不允许给目录创建硬链接。

范例：普通文件再分类

文件类型从 Linux 文件系统那么一个级别分了以上那么多类型，不过普通文件还是可以再分的（根据文件内容的"数据结构"分），比如常见的文本文件，可执行的 `ELF` 文件，`odt` 文档，`jpg` 图片格式，`swap` 分区文件，`pdf` 文件。除了文本文件外，它们大多是二进制文件，有特定的结构，因此需要有专门的工具来创建和编辑它们。关于各类文件的格式，可以参考相关文档标准。不过非常值得深入了解 Linux 下可执行的 `ELF` 文件的工作原理，如果有兴趣，建议阅读一下参考资料中和 `ELF` 文件相关部分，这一部分对于嵌入式 Linux 工程师至关重要。

虽然各类普通文件都有专属的操作工具，但是还是可以直接读、写它们，这里先提到这么几个工具，回头讨论细节。

- `od` ：以八进制或者其他格式"导出"文件内容。
- `strings` ：读出文件中的字符（可打印的字符）
- `gcc` ，`gdb` ，`readelf` ，objdump` 等： `ELF` 文件分析、处理工具（ `gcc` 编译器、 `gdb` 调试器、 `readelf` 分析 `ELF` 文件， `objdump` 反编译工具）

再补充一个非常重要的命令，`file` ，这个命令用来查看各类文件的属性。和 `stat` 命令相比，它可以进一步识别普通文件，即 `stat` 命令显示的 `regular file` 。因为 `regular file` 可以有各种不同的结构，因此在操作系统的支持下得到不同的解释，执行不同的动作。虽然，Linux 下，文件也会加上特定的后缀以便用户能够方便地识别文件的类型，但是 Linux 操作系统根据文件头识别各类文件，而不是文件后缀，这样在解释相应的文件时就更不容易出错。下面简单介绍 `file` 命令的用法。

```
$ file ./
./: directory
$ file /etc/profile
/etc/profile: ASCII English text
$ file /lib/libc-2.5.so
/lib/libc-2.5.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), not str
ipped
$ file /bin/test
/bin/test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linke
d (uses shared libs), stripped
$ file /dev/hda
/dev/hda: block special (3/0)
$ file /dev/console
/dev/console: character special (5/1)
$ cp /etc/profile .
$ tar zcf profile.tar.gz profile
$ file profile.tar.gz
profile.tar.gz: gzip compressed data, from Unix, last modified: Tue Jan  4 18:53:53 20
00
$ mkfifo fifo_test
$ file fifo_test
fifo_test: fifo (named pipe)
```

更多用法见 `file` 命令的手册，关于 `file` 命令的实现原理，请参考 `magic` 的手册（看看 `/etc/file/magic` 文件，了解什么是文件的 `magic number` 等）。

文件属主

Linux 作为一个多用户系统，为多用户使用同一个系统提供了极大的方便，比如对于系统上的文件，它通过属主来区分不同的用户，以便分配它们对不同文件的操作权限。为了更方便地管理，文件属主包括该文件所属用户，以及该文件所属的用户组，因为用户可以属于多个组。先来简单介绍 Linux 下用户和组的管理。

Linux 下提供了一组命令用于管理用户和组，比如用于创建用户的 `useradd` 和 `groupadd`，用于删除用户的 `userdel` 和 `groupdel`，另外，`passwd` 命令用于修改用户密码。当然，Linux 还提供了两个相应的配置，即 `/etc/passwd` 和 `/etc/group`，另外，有些系统还把密码单独放到了配置文件 `/etc/shadow` 中。关于它们的详细用法请参考后面的资料，这里不再介绍，仅介绍文件和用户之间的一些关系。

范例：修改文件的属主

```
$ chown 用户名:组名 文件名
```

如果要递归地修改某个目录下所有文件的属主，可以添加 `-R` 选项。

从本节开头列出的文件结构体中，可以看到仅仅有用户 `ID` 和组 `ID` 的信息，但 `ls -l` 的结果却显示了用户名和组名信息，这个是怎么实现的呢？下面先看看 `-n` 的结果：

范例：查看文件的属主

```
$ ls -n regular_file
-rw-r--r-- 1 0 0 115 2007-12-07 23:45 regular_file
$ ls -l regular_file
-rw-r--r-- 1 root root 115 2007-12-07 23:45 regular_file
```

范例：分析文件属主实现的背后原理

可以看到，`ls -n` 显示了用户 `ID` 和组 `ID`，而 `ls -l` 显示了它们的名字。还记得上面提到的两个配置文件 `/etc/passwd` 和 `/etc/group` 文件么？它们分别存放了用户 `ID` 和用户名，组 `ID` 和组名的对应关系，因此很容易想到 `ls -l` 命令在实现时是如何通过文件结构体的 `ID` 信息找到它们对应的名字信息的。如果想对 `ls -l` 命令的实现有更进一步的了解，可以用 `strace` 跟踪看看它是否读取了这两个配置文件。

```
$ strace -f -o strace.log ls -l regular_file
$ cat strace.log | egrep "passwd|group|shadow"
2989  open("/etc/passwd", O_RDONLY)     = 3
2989  open("/etc/group", O_RDONLY)      = 3
```

说明： `strace` 可以用来跟踪系统调用和信号。如同 `gdb` 等其他强大的工具一样，它基于系统的 `ptrace` 系统调用实现。

实际上，把属主和权限分开介绍不太好，因为只有它们两者结合才使得多用户系统成为可能，否则无法隔离不同用户对某个文件的操作，所以下面来介绍文件操作权限。

文件权限

从 `ls -l` 命令的结果的第一列的后 9 个字符中，可以看到类似这样的信息 `rwxr-xr-x`，它们对应于文件结构体的 `st_mode` 部分（ `st_mode` 包含文件类型信息和文件权限信息两部分）。这类信息可以分成三部分，即 `rwx`，`r-x`，`r-x`，分别对应该文件所属用户、所属组、其他组对该文件的操作权限，如果有 `rwx` 中任何一个表示可读、可写、可执行，如果为 `-` 表示没有这个权限。对应地，可以用八进制来表示它，比如 `rwxr-xr-x` 就可表示成二进制 111101101，对应的八进制则为 755 。正因为如此，要修改文件的操作权限，也可以有多种方式来实现，它们都可通过 `chmod` 命令来修改。

范例：给文件添加读、写、可执行权限

比如，把 `regular_file` 的文件权限修改为所有用户都可读、可写、可执行，即 `rwxrwxrwx`，也可表示为 111111111，翻译成八进制，则为 777 。这样就可以通过两种方式修改这个权限。

```
$ chmod a+rwx regular_file
```

或

```
$ chmod 777 regular_file
```

说明： `a` 指所有用户，如果只想给用户本身可读可写可执行权限，那么可以把 `a` 换成 `u` ；而 `+` 就是添加权限，相反的，如果想去掉某个权限，用 `-` ，而 `rwx` 则对应可读、可写、可执行。更多用法见 `chmod` 命令的帮助。

实际上除了这些权限外，还有两个涉及到安全方面的权限，即 `setuid/setgid` 和只读控制等。

如果设置了文件（程序或者命令）的 `setuid/setgid` 权限，那么用户将可用 `root` 身份去执行该文件，因此，这将可能带来安全隐患；如果设置了文件的只读权限，那么用户将仅仅对该文件将有可读权限，这为避免诸如 `rm -rf` 的“可恶”操作带来一定的庇佑。

范例：授权普通用户执行root所属命令

默认情况下，系统是不允许普通用户执行 `passwd` 命令的，通过 `setuid/setgid` ，可以授权普通用户执行它。

```
$ ls -l /usr/bin/passwd
-rwx--x--x 1 root root 36092 2007-06-19 14:59 /usr/bin/passwd
$ su      #切换到root用户，给程序或者命令添加“粘着位”
$ chmod +s /usr/bin/passwd
$ ls -l /usr/bin/passwd
-rws--s--x 1 root root 36092 2007-06-19 14:59 /usr/bin/passwd
$ exit
$ passwd #普通用户通过执行该命令，修改自己的密码
```

说明：

> `setuid` 和 `setgid` 位是让普通用户可以以 `root` 用户的角色运行只有 `root` 帐号才能运行的程序或命令。

虽然这在一定程度上为管理提供了方便，比如上面的操作让普通用户可以修改自己的帐号，而不是要 `root` 帐号去为每个用户做这些工作。关于 `setuid/setgid` 的更多详细解释，请参考最后推荐的资料。

范例：给重要文件加锁

只读权限示例：给重要文件加锁（添加不可修改位 [immutable])），以避免各种误操作带来的灾难性后果（例如 `:` `rm -rf` ）

```
$ chattr +i regular_file
$ lsattr regular_file
----i-------- regular_file
$ rm regular_file    #加immutable位后就无法对文件进行任何“破坏性”的活动啦
rm: remove write-protected regular file `regular_file'? y
rm: cannot remove `regular_file': Operation not permitted
$ chattr -i regular_file #如果想对它进行常规操作，那么可以把这个位去掉
$ rm regular_file
```

说明： `chattr` 可以用于设置文件的特殊权限，更多用法请参考 `chattr` 的帮助。

文件大小

文件大小对于普通文件而言就是文件内容的大小，而目录作为一个特殊的文件，它存放的内容是以目录结构体组织的各类文件信息，所以目录的大小一般都是固定的，它存放的文件个数自然也就有上限，即它的大小除以文件名的长度。设备文件的“文件大小”则对应设备的主、次设备号，而有名管道文件因为特殊的读写性质，所以大小常是 0 。硬链接（目录文件不能创建硬链接）实质上是原文件的一个完整的拷贝，因此，它的大小就是原文件的大小。而软链接只是一个 `inode` ，存放了一个指向原文件的指针，因此它的大小仅仅是原文件名的字节数。下面我们通过演示增加记忆。

范例：查看普通文件和链接文件

原文件，链接文件文件大小的示例：

```
$ echo -n "abcde" > regular_file   #往regular_file写入5字节
$ ls -l regular_file*
-rw-r--r-- 2 root root  5 2007-12-08 15:28 regular_file
-rw-r--r-- 2 root root  5 2007-12-08 15:28 regular_file_hard_file
lrwxrwxrwx 1 root root 12 2007-12-07 20:15 regular_file_soft_link -> regular_file
lrwxrwxrwx 1 root root 22 2007-12-08 15:21 regular_file_soft_link_link -> regular_file
_soft_link
$ i="regular_file"
$ j="regular_file_soft_link"
$ echo ${#i} ${#j}   #软链接存放的刚好是它们指向的原文件的文件名的字节数
12 22
```

范例：查看设备文件

设备号对应的文件大小：主、次设备号

```
$ ls -l hda1_block_dev_file
brw-r--r-- 1 root root 3, 1 2007-12-07 21:44 hda1_block_dev_file
$ ls -l null_char_dev_file
crw-r--r-- 1 root root 1, 3 2007-12-07 21:43 null_char_dev_file
```

补充：主 `(major)、次` (minor)设备号的作用有不同。当一个设备文件被打开时，内核会根据主设备号（ `major number` ）去查找在内核中已经以主设备号注册的驱动（可以 `cat /proc/devices` 查看已经注册的驱动号和主设备号的对应情况），而次设备号（ `minor number` ）则是通过内核传递给了驱动本身（参考《The Linux Primer》第十章）。因此，对于内核而言，通过主设备号就可以找到对应的驱动去识别某个设备，而对于驱动而言，为了能够更复杂地访问设备，比如访问设备的不同部分（如硬件通过分区分成不同部分，而出现 `hda1` ， `hda2` ， `hda3` 等），比如产生不同要求的随机数（如 `/dev/random` 和 `/dev/urandom` 等）。

范例：查看目录

目录文件的大小，为什么是这样呢？看看下面的目录结构体的大小，目录文件的 Block 中存放了该目录下所有文件名的入口。

```
$ ls -ld directory_file/
drwxr-xr-x 2 root root 4096 2007-12-07 23:14 directory_file/
```

目录的结构体如下：

```
struct dirent {
    long d_ino;
    off_t d_off;
    unsigned short d_reclen;
    char d_name[NAME_MAX+1]; /* 文件名称 */
}
```

文件访问、更新、修改时间

文件的时间属性可以记录用户对文件的操作信息，在系统管理、判断文件版本信息等情况下将为管理员提供参考。因此，在阅读文件时，建议用 `cat` 等阅读工具，不要用编辑工具 `vim` 去阅读，因为即使没有做任何修改操作，一旦执行了保存命令，将修改文件的时间戳信息。

文件名

文件名并没有存放在文件结构体内，而是存放在它所在的目录结构体中。所以，在目录的同一级别中，文件名必须是唯一的。

文件的基本操作

对于文件，常见的操作包括创建、删除、修改、读、写等。关于各种操作对应的“背后动作”将在下一章《Shell编程范例之文件系统操作》详细分析。

范例：创建文件

`socket` 文件是一类特殊的文件，可以通过 C 语言创建，这里不做介绍（暂时不知道是否可以用命令直接创建），其他文件将通过命令创建。

```
$ touch regular_file      #创建普通文件
$ mkdir directory_file     #创建目录文件，目录文件里头可以包含更多文件
$ ln regular_file regular_file_hard_link  #硬链接，是原文件的一个完整拷比
$ ln -s regular_file regular_file_soft_link  #类似一个文件指针，指向原文件
$ mkfifo fifo_pipe   #或者通过 "mknod fifo_pipe p" 来创建，FIFO满足先进先出的特点
$ mknod hda1_block_dev_file b 3 1  #块设备
$ mknod null_char_dev_file c 1 3   #字符设备
```

创建一个文件实际上是在文件系统中添加了一个节点（`inode)，该节点信息将保存到文件系统的节点表中。更形象地说，就是在一颗树上长了一颗新的叶子（文件）或者枝条（目录文件，上面还可以长叶子的那种），这些可以通过 tree`命令或者 ls` 命令形象地呈现出来。文件系统从日常使用的角度，完全可以当成一颗倒立的树来看，因为它们太像了，太容易记忆啦。

```
$ tree 当前目录
```

或者

```
$ ls 当前目录
```

范例：删除文件

删除文件最直接的印象是这个文件再也不存在了，这同样可以通过 `ls` 或者 `tree` 命令呈现出来，就像树木被砍掉一个分支或者摘掉一片叶子一样。实际上，这些文件删除之后，并不是立即消失了，而是仅仅做了删除标记，因此，如果删除之后，没有相关的磁盘写操作把相应的磁盘空间“覆盖”，那么原理上是可以恢复的（虽然如此，但是这样的工作往往很麻烦，所以在删除一些重要数据时，请务必三思而后行，比如做好备份工作），相应的做法可以参考后续资料。

具体删除文件的命令有 `rm` ，如果要删除空目录，可以用 `rmdir` 命令。例如：

```
$ rm regular_file
$ rmdir directory_file
$ rm -r directory_file_not_empty
```

`rm` 有两个非常重要的参数，一个是 `-f` ，这个命令是非常“野蛮的”，它估计给很多 Linux user 带来了痛苦，另外一个是 `-i` ，这个命令是非常“温柔的”，它估计让很多用户感觉烦躁不已。用哪个还是根据您的“心情”吧，如果做好了充分的备份工作，或者采取了一些有效避免灾难性后果的动作的话，您在做这些工作的时候就可以放心一些啦。

范例：复制文件

文件的复制通常是指文件内容的“临时”复制。通过这一节开头的介绍，我们应该了解到，文件的硬链接和软链接在某种意义上说也是“文件的复制”，前者同步复制文件内容，后者在读写的情况下同步“复制”文件内容。例如：

用 `cp` 命令常规地复制文件（复制目录需要 `-r` 选项）

```
$ cp regular_file regular_file_copy
$ cp -r diretory_file directory_file_copy
```

创建硬链接（ `link` 和 `copy` 不同之处是： `link` 为同步更新， `copy` 则不然，复制之后两者不再相关）

```
$ ln regular_file regular_file_hard_link
```

创建软链接

```
$ ln -s regular_file regluar_file_soft_link
```

范例：修改文件名

修改文件名实际上仅仅修改了文件名标识符。可以通过 `mv` 命令来实现修改文件名操作（即重命名）。

```
$ mv regular_file regular_file_new_name
```

范例：编辑文件

编辑文件实际上是操作文件的内容，对应普通文本文件的编辑，这里主要涉及到文件内容的读、写、追加、删除等。这些工作通常会通过专门的编辑器来做，这类编辑器有命令行下的 `vim` 、 `emacs` 和图形界面下的 `gedit,kedit` 等。如果是一些特定的文件，会有专门的编辑和处理工具，比如图像处理软件 `gimp` ，文档编辑软件 `OpenOffice` 等。这些工具一般都会有专门的教程。

下面主要简单介绍 Linux 下通过重定向来实现文件的这些常规的编辑操作。

创建一个文件并写入 `abcde`

```
$ echo "abcde" > new_regular_file
```

再往上面的文件中追加一行 `abcde`

```
$ echo "abcde" >> new_regular_file
```

按行读一个文件

```
$ while read LINE; do echo $LINE; done < test.sh
```

提示：如果要把包含重定向的字符串变量当作命令来执行，请使用 `eval` 命令，否则无法解释重定向。例如，

```
$ redirect="echo \"abcde\" >test_redirect_file"
$ $redirect   #这里会把>当作字符 > 打印出来，而不会当作 重定向 解释
"abcde" >test_redirect_file
$ eval $redirect    #这样才会把 > 解释成 重定向
$ cat test_redirect_file
abcde
```

范例：压缩／解压缩文件

压缩和解压缩文件在一定意义上来说是为了方便文件内容的传输，不过也可能有一些特定的用途，比如内核和文件系统的映像文件等（更多相关的知识请参考后续资料）。

这里仅介绍几种常见的压缩和解压缩方法：

tar

```
$ tar -cf file.tar file   #压缩
$ tar -xf file.tar    #解压
```

gz

```
$ gzip  -9 file
$ gunzip file
```

tar.gz

```
$ tar -zcf file.tar.gz file
$ tar -zxf file.tar.gz
```

bz2

```
$ bzip2 file
$ bunzip2 file
```

tar.bz2

```
$ tar -jcf file.tar.bz2 file
$ tar -jxf file.tar.bz2
```

通过上面的演示，应该已经非常清楚 `tar` ， `bzip2，` bunzip2， `gzip，` gunzip `命令的角色了吧？如果还不清楚，多操作和比较一些上面的命令，并查看它们的手册：` man tar`...

范例：文件搜索（文件定位）

文件搜索是指在某个目录层次中找出具有某些属性的文件在文件系统中的位置，这个位置如果扩展到整个网络，那么可以表示为一个 `URL` 地址，对于本地的地址，可以表示为 `file://+` 本地路径。本地路径在 Linux 系统下是以 `/` 开头，例如，每个用户的家目录可以表示为： `file:///home/` 。下面仅仅介绍本地文件搜索的一些办法。

`find` 命令提供了一种“及时的”搜索办法，它根据用户的请求，在指定的目录层次中遍历所有文件直到找到需要的文件为止。而 `updatedb+locate` 提供了一种“快速的”的搜索策略， `updatedb` 更新并产生一个本地文件数据库，而 `locate` 通过文件名检索这个数据库以便快速找到相应的文件。前者支持通过各种文件属性进行搜索，并且提供了一个接口（ `-exec` 选项）用于处理搜索后的文件。因此为“单条命令”脚本的爱好者提供了极大的方便，不过对于根据文件名的搜索而言， `updatedb+locate` 的方式在搜索效率上会有明显提高。下面简单介绍这两种方法：

`find` 命令基本使用演示

```
$ find ./ -name "*.c" -o -name "*.h"  #找出所有的C语言文件，-o是或者
$ find ./ \( -name "*.c" -o -name "*.h" \) -exec mv '{}' ./c_files/ \;
# 把找到的文件移到c_files下，这种用法非常有趣
```

上面的用法可以用 `xargs` 命令替代

```
$ find ./ -name "*.c" -o -name "*.h" | xargs -i mv '{}' ./c_files/
# 如果要对文件做更复杂的操作，可以考虑把mv改写为你自己的处理命令，例如，我需要修
```

改所有的文件名后缀为大写。

```
$ find ./ -name "*.c" -o -name "*.h" | xargs -i ./toupper.sh '{}' ./c_files/
```

`toupper.sh` 就是我们需要实现的转换小写为大写的一个处理文件，具体实现如下：

```
$ cat toupper.sh
#!/bin/bash

# the {} will be expended to the current line and becomen the first argument of this s
cript
FROM=$1
BASENAME=${FROM##*/}

BASE=${BASENAME%.*}
SUFFIX=${BASENAME##*.}

TOSUFFIX="$(echo $SUFFIX | tr '[a-z]' '[A-Z]')"
TO=$2/$BASE.$TOSUFFIX
COM="mv $FROM $TO"
echo $COM
eval $COM
```

`updatedb+locate` 基本使用演示

```
$ updatedb #更新库
$ locate find*.gz #查找包含find字符串的所有gz压缩包
```

实际上，除了上面两种命令外，Linux 下还有命令查找工具：`which` 和 `whereis` ，前者用于返回某个命令的全路径，而后者用于返回某个命令、源文件、`man 文件的路径。例如，查找` find` 命令的绝对路径：

```
$ which find
/usr/bin/find
$ whereis find
find: /usr/bin/find /usr/X11R6/bin/find /usr/bin/X11/find /usr/X11/bin/find /usr/man/m
an1/find.1.gz /usr/share/man/man1/find.1.gz /usr/X11/man/man1/find.1.gz
```

需要提到的是，如果想根据文件的内容搜索文件，那么 `find` 和 `updatedb+locate` 以及 `which` ， `whereis` 都无能为力啦，可选的方法是 `grep` ， `sed` 等命令，前者在加上 `-r` 参数以后可以在指定目录下文件中搜索指定的文件内容，后者再使用 `-i` 参数后，可以对文件内容进行替换。它们的基本用法在前面的章节中已经详细介绍了，这里就不再赘述。

值得强调的是，这些命令对文件的操作非常有意义。它们在某个程度上把文件系统结构给抽象了，使得对整个文件系统的操作简化为对单个文件的操作，而单个文件如果仅仅考虑文本部分，那么最终却转化成了之前的字符串操作，即上一节讨论过的内容。为了更清楚地了解文件的组织结构，文件之间的关系，在下一节将深入探讨文件系统。

参考资料

- 从文件 I/O 看 Linux 的虚拟文件系统
- Linux 文件系统剖析
- 《Linux 核心》第九章 文件系统
- Linux Device Drivers, 3rd Edition
- 技巧：Linux I/O 重定向的一些小技巧
- Intel 平台下 Linux 中 ELF 文件动态链接的加载、解析及实例分析:
    - part1,
    - part2
- Shell 脚本调试技术
- ELF 文件格式及程序加载执行过程总汇
- Linux下 C 语言编程——文件的操作
- "Linux下 C 语言编程" 的 文件操作部分
- Filesystem Hierarchy Standard
- 学会恢复 Linux系统里被删除的 Ext3 文件
- 使用mc恢复被删除文件
- linux ext3 误删除及恢复原理
- Linux压缩／解压缩方式大全
- Everything is a byte

后记

- 考虑到文件和文件系统的重要性，将把它分成三个小节来介绍：文件、文件系统、程序与进程。在“文件”这一部分，主要介绍文件的基本属性和常规操作，在“文件系统”那部分，将深入探讨 Linux 文件系统的各个部分（包括 Linux 文件系统的结构、具体某个文件系统的大体结构分析、底层驱动的工作原理），在“程序与进程”一节将专门讨论可执行文件的相关内容（包括不同的程序类型、加载执行过程、不同进程之间的交互[命令管道和无名管道、信号通信]、对进程的控制等）

- 有必要讨论清楚 目录大小 的含义，另外，最好把一些常规的文件操作全部考虑到，包括文件的读、写、执行、删除、修改、复制、压缩／解压缩等
- 下午刚从上海回来，比赛结果很“糟糕”，不过到现在已经不重要了，关键是通过决赛发现了很多不足，发现了设计在系统开发中的关键角色，并且发现了上海是个美丽的城市，上交也是个美丽的大学。回来就开始整理这个因为比赛落下了两周的 Blog
- 12月15日，添加文件搜索部分内容

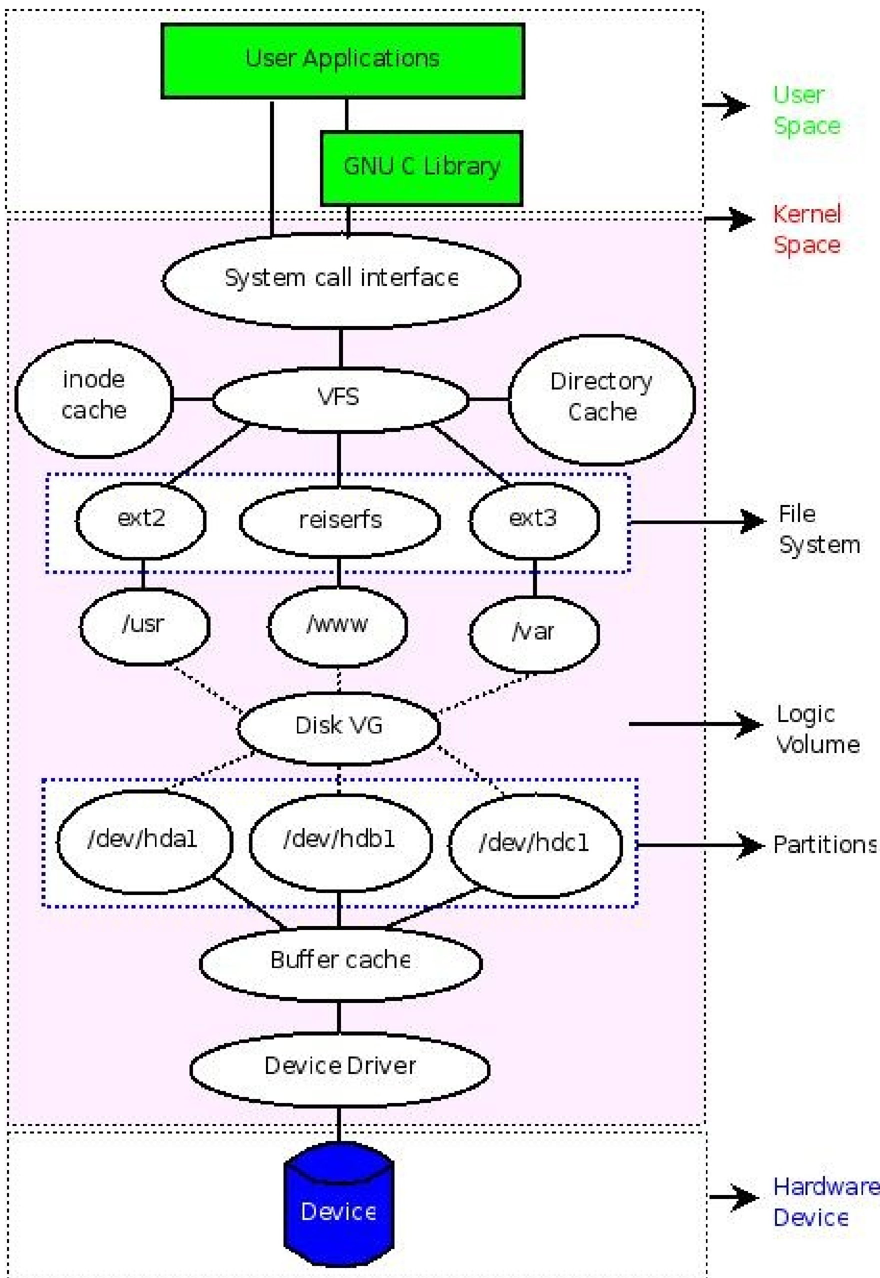
文件系统操作

前言

准备了很久，找了好多天资料，还不知道应该如何动笔写：因为担心拿捏不住，所以一方面继续查找资料，一方面思考如何来写。作为《Shell编程范例》的一部分，希望它能够很好地帮助 Shell 程序员理解如何用 Shell 命令来完成和 Linux 系统关系非常大的文件系统的各种操作，希望让 Shell 程序员中对文件系统"混沌"的状态从此消失，希望文件系统以一种更为清晰的样子呈现在眼前。

文件系统在 Linux 操作系统中的位置

如何来认识文件系统呢？从 Shell 程序员的角度来看，文件系统就是一个用来组织各种文件的方法。但是文件系统无法独立于硬件存储设备和操作系统而存在，因此还是有必要来弄清楚硬件存储设备、分区、操作系统、逻辑卷、文件系统等各种概念之间的联系，以便理解文件系统常规操作的一些“细节”。这个联系或许（也许会有一些问题）可以通过这样一种方式来呈现：



Filesystem Structure in Linux Operating System

从图中可以清晰地看到各个“概念”之间的关系，它们以不同层次分布，覆盖硬件设备、系统内核空间、系统用户空间。在用户空间，用户可以不管内核如何操作具体硬件设备，仅仅使用程序员设计的各种界面就可以，而普通程序员也仅仅需要利用内核提供的各种接口（System

Call）或者一些C库来和内核进行交互，而无须关心具体的实现细节。不过对于操作系统开发人员，他们需要在内核空间设计特定的数据结构来管理和组织底层的硬件设备。

下面从下到上的方式（即从底层硬件开始），用工具来分析和理解图中几个重要概念。（如果有兴趣，可以先看看下面的几则资料）

参考资料：

- 从文件 I/O 看 Linux 的虚拟文件系统
- Linux 文件系统剖析
- 第九章 文件系统
- Linux 逻辑盘卷管理 LVM 详解

硬件管理和设备驱动

Linux 系统通过设备驱动管理硬件设备。如果添加了新的硬件设备，那么需要编写相应的硬件驱动来管理它。对于一些常见的硬件设备，系统已经自带了相应的驱动，编译内核时，选中它们，然后编译成内核的一部分或者以模块的方式编译。如果以模块的方式编译，那么可以在系统的 `/lib/modules/$(uname -r)` 目录下找到对应的模块文件。

范例：查找设备所需的驱动文件

比如，可以这样找到相应的 scsi 驱动和 usb 驱动对应的模块文件：

更新系统中文件索引数据库(有点慢）

```
$ updatedb
```

查找 scsi 相关的驱动

```
$ locate scsi*.ko
```

查找 usb 相关的驱动

```
$ locate usb*.ko
```

这些驱动以 `.ko` 为后缀，在安装系统时默认编译为了模块。实际上可以把它们编译为内核的一部分，仅仅需要在编译内核时选择为 `[*]` 即可。但是，很多情况下会以模块的方式编译它们，这样可以减少内核的大小，并根据需要灵活地加载和卸载它们。下面简单地演示如何卸载模块、加载模块以及查看已加载模块的状态。

可通过 `/proc` 文件系统的 `modules` 文件检查内核中已加载的各个模块的状态，也可以通过 `lsmod` 命令直接查看它们。

```
$ cat /proc/modules
```

或者

```
$ lsmod
```

范例：查看已经加载的设备驱动

查看 scsi 和 usb 相关驱动，结果各列为模块名、模块大小、被其他模块的引用情况（引用次数、引用它们的模块）

```
$ lsmod | egrep "scsi|usb"
usbhid                  29536  0
hid                     28928  1 usbhid
usbcore                138632  4 usbhid,ehci_hcd,ohci_hcd
scsi_mod               147084  4 sg,sr_mod,sd_mod,libata
```

范例：卸载设备驱动

下面卸载 `usbhid` 模块看看（不要卸载scsi的驱动！因为你的系统可能就跑在上面，如果确实想玩玩，卸载前记得保存数据），通过 `rmmod` 命令就可以实现，先切换到 Root 用户：

```
$ sudo -s
# rmmod usbhid
```

再查看该模块的信息，已经看不到了吧

```
$ lsmod | grep ^usbhid
```

范例：挂载设备驱动

如果有个 usb 鼠标，那么移动一下，是不是发现动不了啦？因为设备驱动都没有了，设备自然就没法用罗。不过不要紧张，既然知道原因，那么重新加载驱动就可以，下面用 `insmod` 把 `usbhid` 模块重新加载上。

```
$ sudo -s
# insmod `locate usbhid.ko`
```

`locate usbhid.ko` 是为了找出 `usbhid.ko` 模块的路径，如果之前没有 `updatedb`，估计用它是找不到了，不过也可以直接到 `/lib/modules` 目录下用 `find` 把 `usbhid.ko` 文件找到。

```
# insmod $(find /lib/modules -name "*usbhid.ko*" | grep `uname -r`)
```

现在鼠标又可以用啦，不信再动一下鼠标 :-)

到这里，硬件设备和设备驱动之间关系应该是比较清楚了。如果没有，那么继续下面的内容。

范例：查看设备驱动对应的设备文件

Linux 设备驱动关联着相应的设备文件，而设备文件则和硬件设备一一对应。这些设备文件都统一存放在系统的 `/dev/` 目录下。

例如，scsi 设备对应 `/dev/sda`，`/dev/sda1`，`/dev/sda2`... 下面查看这些设备信息。

```
$ ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 2007-12-28 22:49 /dev/sda
brw-rw---- 1 root disk 8, 1 2007-12-28 22:50 /dev/sda1
brw-rw---- 1 root disk 8, 3 2007-12-28 22:49 /dev/sda3
brw-rw---- 1 root disk 8, 4 2007-12-28 22:49 /dev/sda4
brw-rw---- 1 root disk 8, 5 2007-12-28 22:50 /dev/sda5
brw-rw---- 1 root disk 8, 6 2007-12-28 22:50 /dev/sda6
brw-rw---- 1 root disk 8, 7 2007-12-28 22:50 /dev/sda7
brw-rw---- 1 root disk 8, 8 2007-12-28 22:50 /dev/sda8
```

可以看到第一列第一个字符都是 `b`，第五列都是数字 8 。 `b` 表示该文件是一个块设备文件，对应地，如果是 `c` 则表示字符设备（例如 `/dev/ttyS0)，关于块设备和字符设备的区别，可以看这里：

> - 字符设备：字符设备就是能够像字节流一样访问的设备，字符终端和串口就属于字符设备。
> - 块设备：块设备上可以容纳文件系统。与字符设备不同，在读写时，块设备每次只能传输一个或多个完整的块。在 Linux 操作系统中，应用程序可以像访问字符设备一样读写块设备（一次读取或写入任意的字节数据）。因此，块设备和字符设备的区别仅仅是在内核中对于数据的管理不同。

数字 8 则是该硬件设备在内核中对应的设备编号，可以在内核的 `Documentation/devices.txt` 和 `/proc/devices` 文件中找到设备号分配情况。但是为什么同一个设备会对应不同的设备文件（ `/dev/sda` 后面为什么还有不同的数字，而且 `ls` 结果中的第 6 列和它们对应起来）。这实际上是为了区分不同设备的不同部分。对于硬盘，这样可以处理硬盘内部的不同分区。就内核而言，它仅仅需要通过第 5 列的设备号就可以找到对应的硬件设备，但是对于驱动模

块来说，它还需要知道如何处理不同的分区，于是就多了一个辅设备号，即第 6 列对应的内容。这样一个设备就有了主设备号（第 5 列）和辅设备号（第 6 列），从而方便地实现对各种硬件设备的管理。

因为设备文件和硬件是对应的，这样可以直接从 `/dev/sda` （如果是 `IDE` 的硬盘，那么对应的设备就是 `/dev/hda` 啦）设备中读出硬盘的信息，例如：

范例：访问设备文件

用 `dd` 命令复制出硬盘的前 512 个字节，要 Root 用户

```
$ sudo dd if=/dev/sda of=mbr.bin bs=512 count=1
```

用 `file` 命令查看相应的信息

```
$ file mbr.bin
mbr.bin: x86 boot sector, LInux i386 boot LOader; partition 3: ID=0x82, starthead 254,
 startsector 19535040, 1959930 sectors; partition 4: ID=0x5, starthead 254, startsecto
r 21494970, 56661255 sectors, code offset 0x48
```

也可以用 `od` 命令以 16 进制的形式读取并进行分析

```
$ od -x mbr.bin
```

`bs` 是块的大小（以字节 `bytes` 为单位），`count` 是块数

因为这些信息并不直观（而且下面会进一步深入分析），那么先来看看另外一个设备文件，将可以非常直观地演示设备文件和硬件的对应关系。还是以鼠标为例吧，下面来读取鼠标对应的设备文件的信息。

```
$ sudo -s
# cat /dev/input/mouse1 | od -x
```

你的鼠标驱动可能不太一样，所以设备文件可能是其他的，但是都会在 `/dev/input` 下。

移动鼠标看看，是不是发现有不同信息输出。基于这一原理，我们经常通过在一端读取设备文件 `/dev/ttyS0` 中的内容，而在另一端往设备文件 `/dev/ttyS0` 中写入内容来检查串口线是否被损坏。

到这里，对设备驱动、设备文件和硬件设备之间的关联应该是印象更深刻了。如果想深入了解设备驱动的工作原理和设备驱动的编写，那么看看下面列出的相关资料，开始设备驱动的编写历程吧。

参考资料：

- Compile linux kernel 2.6
- Linux 系统的硬件驱动程序编写原理
- Linux 下 USB设备的原理、配置、常见问题
- The Linux Kernel Module Programming Guide
- Linux 设备驱动开发

理解、查看磁盘分区

实际上内存、u 盘等都可以作为文件系统底层的“存储”设备，但是这里仅用硬盘作为实例来介绍磁盘和分区的关系。

目前 Linux 的分区依然采用第一台PC硬盘所使用的分区原理，下面逐步分析和演示这一分区原理。

磁盘分区基本原理

先来看看几个概念：

- 设备管理和分区

  Linux 下，每一个存储设备对应一个系统的设备文件，对于硬盘等 `IDE` 和 `SCSI` 设备，在系统的 `/dev` 目录下可以找到对应的包含字符 `hd` 和 `sd` 的设备文件。而根据硬盘连接的主板设备接口和数据线接口的不同，在 `hd` 或者 `sd` 字符后面可以添加一个从 `a` 到 `z` 的字符，例如 `hda` ， `hdb` ， `hdc` 和 `sda` ， `sdb` ， `sdc` 等，另外为了区别同一个硬件设备的不同分区，在后面还可以添加了一个数字，例如 `hda1` ， `hda2` ， `hda3` 和 `sda1` ， `sda2` ， `sda3` ，所以在 `/dev` 目录下，可以看到很多类似的设备文件。

- 各分区的作用

在分区时常遇到主分区和逻辑分区的问题，这实际上是为了方便扩展分区，正如后面的逻辑卷的引入是为了更好地管理多个硬盘一样，引入主分区和逻辑分区可以方便地进行分区的管理。

Linux 系统中每一个硬盘设备最多由 4 个主分区（包括扩展分区）构成。

主分区的作用是计算机用来进行启动操作系统的，因此每一个操作系统的启动程序或者称作是引导程序，都应该存放在主分区上。 Linux 规定主分区（或者扩展分区）占用分区编号中的前 4 个。所以会看到主分区对应的设备文件为 `/dev/hda1-4` 或者 `/dev/sda1-4` ，而不会是 `hda5` 或者 `sda5` 。

扩展分区则是为了扩展更多的逻辑分区的，在 Linux 下，逻辑分区占用了 `hda5-16` 或者 `sda5-16` 等 12 个编号。

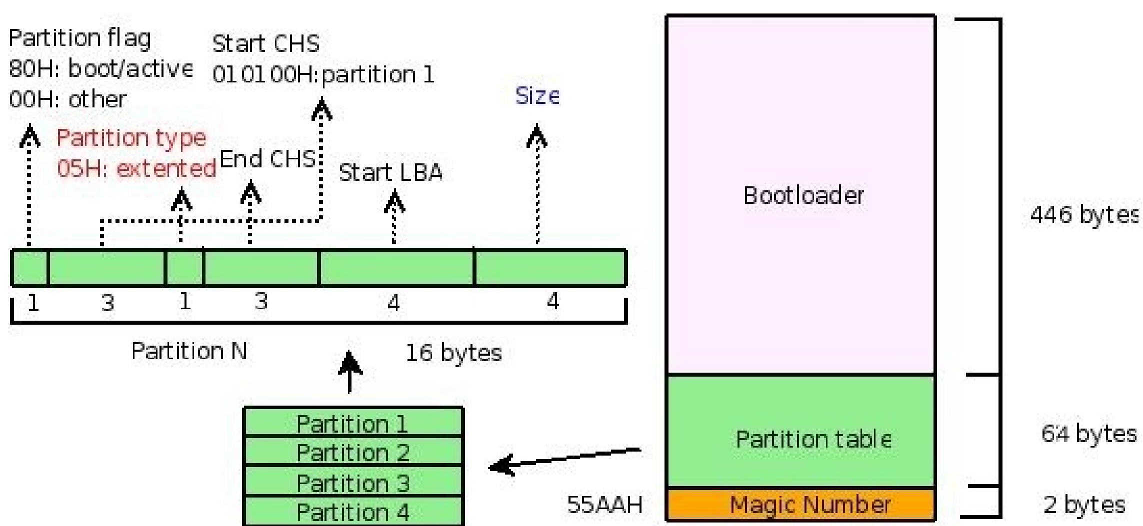
- 分区类型

它规定了这个分区上的文件系统的类型。Linux支持诸如msdoc,vfat,ext2,ext3等诸多的文件系统类型，更多信息在下一小节进行进一步的介绍。

通过分析 MBR 来理解分区原理

下面通过分析硬盘的前 512 个字节（即 `MBR` ）来分析和理解分区。

先来看看这张图：



它用来描述 `MBR` 的结构。 `MBR` 包括引导部分、分区表、以及结束标记`(55AAH)，分别占用了 512 字节中 446 字节、 64 字节和 2 字节。这里仅仅关注分区表部分，即中间的 64 字节以及图中左边的部分。

由于我用的是 `SCSI` 的硬盘，下面从 `/dev/sda` 设备中把硬盘的前 512 个字节拷贝到文件 `mbr.bin` 中。

```
$ sudo -s
# dd if=/dev/sda of=mbr.bin bs=512 count=1
```

下面用 `file` ， `od` ， `fdisk` 等命令来分析这段 `MBR` 的数据，并对照上图以便加深理解。

```
$ file mbr.bin
mbr.bin: x86 boot sector, LInux i386 boot LOader; partition 3: ID=0x82, starthead 254,
 startsector 19535040, 1959930 sectors; partition 4: ID=0x5, starthead 254, startsecto
r 21494970, 56661255 sectors, code offset 0x48
$ od -x mbr.bin | tail -6   #仅关注中间的64字节，所以截取了结果中后6行
0000660 0000 0000 0000 0000 a666 a666 0000 0180
0000700 0001 fe83 ffff 003f 0000 1481 012a 0000
0000720 0000 0000 0000 0000 0000 0000 0000 fe00
0000740 ffff fe82 ffff 14c0 012a e7fa 001d fe00
0000760 ffff fe05 ffff fcba 0147 9507 0360 aa55
$ sudo -s
# fdisk -l | grep ^/  #仅分析MBR相关的部分，不分析逻辑分区部分
/dev/sda1   *           1        1216     9767488+  83  Linux
/dev/sda3            1217        1338      979965   82  Linux swap / Solaris
/dev/sda4            1339        4865    28330627+   5  Extended
```

`file` 命令的结果显示，刚拷贝的 512 字节是启动扇区，用分号分开的几个部分分别是 `bootloader` ，分区 3 和分区 4 。分区 3 的类型是 82，即 `swap` 分区（可以通过 `fdisk` 命令的 `l` 命令列出相关信息），它对应 `fdisk` 的结果中 `/dev/sda3` 所在行的第 5 列，分区 3 的扇区数是 1959930，转换成字节数是 `1959930\*512` （目前，硬盘的默认扇区大小是 512 字节），而 `swap` 分区的默认块大小是 1024 字节，这样块数就是 `:`

```
$ echo 1959930*512/1024 | bc
979965
```

正好是 `fdisk` 结果中 `/dev/sda3` 所在行的第四列对应的块数，同样地，可以对照 `fdisk` 和 `file` 的结果分析分区 4 。

再来看看 `od` 命令以十六进制显示的结果，同样考虑分区 3，计算一下发现，分区 3 对应的 `od` 命令的结果为：

```
fe00 ffff fe82 ffff 14c0 012a e7fa 001d
```

首先是分区标记， `00H` ，从上图中，看出它就不是引导分区（ `80H` 标记的才是引导分区），而分区类型呢？为 `82H` ，和 `file` 显示结果一致，现在再来关注一下分区大小，即 `file` 结果中的扇区数。

```
$ echo "ibase=10;obase=16;1959930" | bc
1DE7FA
```

刚好对应 `e7fa 001d` ，同样地考虑引导分区的结果：

> 0180 0001 fe83 ffff 003f 0000 1481 012a

分区标记： `80H` ，正好反应了这个分区是引导分区，随后是引导分区所在的磁盘扇区情况，010100，即 1 面 0 道 1 扇区。其他内容可以对照分析。

考虑到时间关系，更多细节请参考下面的资料或者查看看系统的相关手册。

补充：安装系统时，可以用 `fdisk` ， `cfdisk` 等命令进行分区。如果要想从某个分区启动，那么需要打上 `80H` 标记，例如可通过 `cfdisk` 把某个分区设置为 `bootable` 来实现。

参考资料：

- Inside the linux boot process
- Develop your own OS: booting
- Redhat9 磁盘分区简介
- Linux partition HOWTO

分区和文件系统的关系

在没有引入逻辑卷之前，分区类型和文件系统类型几乎可以同等对待，设置分区类型的过程就是格式化分区，建立相应的文件系统类型的过程。

下面主要介绍如何建立分区和文件系统类型的联系，即如何格式化分区为指定的文件系统类型。

常见分区类型

先来看看 Linux 下文件系统的常见类型（如果要查看所有 Linux 支持的文件类型，可以用 `fdisk` 命令的 `l` 命令查看，或者通过 `man fs` 查看，也可通过 `/proc/filesystems` 查看到当前内核支持的文件系统类型）

- `ext2` ， `ext3` ， `ext4` ：这三个是 Linux 根文件系统通常采用的类型
- `swap` ：这个是实现 Linux 虚拟内存时采用的一种文件系统，安装时一般需要建立一个专门的分区，并格式化为 `swap` 文件系统（如果想添加更多 `swap` 分区，可以参考本节的参考资料，熟悉 `dd` ， `mkswap` ， `swapon` ， `swapoff` 等命令的用法）
- `proc` ：这是一种比较特别的文件系统，作为内核和用户之间的一个接口存在，建立在内存中（可以通过 `cat` 命令查看 `/proc` 系统下的文件，甚至可以通过修改 `/proc/sys` 下的文件实时调整内核配置，当前前提是需要把 `proc` 文件系统挂载上： `mount -t proc proc /proc`

除了上述文件系统类型外，Linux 支持包括 `vfat` ， `iso` ， `xfs` ， `nfs` 在内各种常见的文件系统类型，在 Linux 下，可以自由地查看和操作 Windows 等其他操作系统使用的文件系统。

那么如何建立磁盘和这些文件系统类型的关联呢？格式化。

格式化的过程实际上就是重新组织分区的过程，可通过 `mkfs` 命令来实现，当然也可以通过 `fdisk` 等命令来实现。这里仅介绍 `mkfs` ， `mkfs` 可用来对一个已有的分区进行格式化，不能实现分区操作（如果要对一个磁盘进行分区和格式化，那么可以用 `fdisk` ）。格式化后，相应分区上的数据就会通过某种特别的文件系统类型进行组织。

范例：格式化文件系统

例如：把 `/dev/sda9` 分区格式化为 `ext3` 的文件系统。

```
$ sudo -s
# mkfs -t ext3 /dev/sda9
```

如果要列出各个分区的文件系统类型，那么可以用 `fdisk -l` 命令。

更多信息请参考下列资料。

参考资料：

- Linux 下加载 swap 分区的步骤
- Linux 下 ISO 镜像文件的制作与刻录
- RAM 磁盘分区解释: [1], [2]
- 高级文件系统实现者指南

分区、逻辑卷和文件系统的关系

上一节直接把分区格式化为某种文件系统类型，但是考虑到扩展新的存储设备的需要，开发人员在文件系统和分区之间引入了逻辑卷。考虑到时间关系，这里不再详述，请参考资料：Linux 逻辑卷管理详解

文件系统的可视化结构

文件系统最终呈现出来的是一种可视化的结构，可用ls,find,tree等命令把它呈现出来。它就像一颗倒挂的“树”，在树的节点上还可以挂载新的“树”。

下面简单介绍文件系统的挂载。

一个文件系统可以通过一个设备挂载（ `mount` ）到某个目录下，这个目录被称为挂载点。有趣的是，在 Linux 下，一个目录本身还可以挂载到另外一个目录下，一个格式化了的文件也可以通过一个特殊的设备 `/dev/loop` 进行挂载（如 `iso` 文件）。另外，就文件系统而言，Linux 不仅支持本地文件系统，还支持远程文件系统（如 `nfs` ）。

范例：挂载文件系统

下面简单介绍文件系统挂载的几个实例。

- 根文件系统的挂载

挂载需要 Root 权限，例如，挂载系统根文件系统 `/dev/sda1` 到 `/mnt`

```
$ sudo -s
# mount -t ext3 /dev/sda1 /mnt/
```

查看 `/dev/sda1` 的挂载情况，可以看到，一个设备可以多次挂载

```
$ mount | grep sda1
/dev/sda1 on / type ext3 (rw,errors=remount-ro)
/dev/sda1 on /mnt type ext3 (rw)
```

对于一个已经挂载的文件系统，为支持不同属性可以重新挂载

```
$ mount -n -o remount, rw /
```

- 挂载一个新增设备

如果内核已经支持 USB 接口，那么插入 u 盘时，可以通过 `dmesg` 命令查看对应的设备号，并挂载它。

查看 `dmesg` 结果中的最后几行内容，找到类似 `/dev/sdN` 的信息，找出 u 盘对应的设备号

```
$ dmesg
```

这里假设 u 盘是 `vfat` 格式，以便在一些打印店里的 Windows 上也可使用

```
# mount -t vfat /dev/sdN /path/to/mountpoint_directory
```

- 挂载一个 iso 文件或者是光盘

对于一些iso文件或者是 iso 格式的光盘，同样可以通过 `mount` 命令挂载。

对于 iso 文件：

```
# mount -t iso9660 /path/to/isofile /path/to/mountpoint_directory
```

对于光盘：

```
# mount -t iso9660 /dev/cdrom /path/to/mountpoint_directory
```

- 挂载一个远程文件系统

```
# mount -t nfs remote_ip:/path/to/share_directory /path/to/local_directory
```

- 挂载一个 proc 文件系统

```
# mount -t proc proc /proc
```

`proc` 文件系统组织在内存中，但是可以把它挂载到某个目录下。通常把它挂载在 `/proc` 目录下，以便一些系统管理和配置工具使用它。例如 `top` 命令用它分析内存的使用情况（读取 `/proc/meminfo` 和 `/proc/stat` 等文件中的内容）；`lsmod` 命令通过它获取内核模块的状态（读取 `/proc/modules` ）；`netstat` 命令通过它获取网络的状态（读取 `/proc/net/dev` 等文件）。当然，也可以编写相关工具。除此之外，通过调整 `/proc/sys` 目录下的文件，可以动态地调整系统配置，比如往 `/proc/sys/net/ipv4/ip_forward` 文件中写入数字 1 就可以让内核支持数据包转发。（更多信息请参考 `proc` 的帮助，`man proc`）

- 挂载一个目录

```
$ mount --bind /path/to/needtomount_directory /path/to/mountpoint_directory
```

这个非常有意思，比如可以把某个目录挂载到 ftp 服务的根目录下，而无须把内容复制过去，就可以把相应目录中的资源提供给别人共享。

范例：卸载某个分区

以上都只提到了挂载，那怎么卸载呢？用 `umount` 命令跟上挂载的源地址或者挂载点（设备，文件，远程目录等）就可以。例如：

```
$ umount /path/to/mountpoint_directory
```

或者

```
$ umount /path/to/mount_source
```

如果想管理大量的或者经常性的挂载服务，那么每次手动挂载是很糟糕的事情。这时就可利用 `mount` 的配置文件 `/etc/fstab`，把 `mount` 对应的参数写到 `/etc/fstab` 文件对应的列中即可实现批量挂载（ `mount -a` ）和卸载（ `umount -a` ）。 `/etc/fstab` 中各列分别为文

件系统、挂载点、类型、相关选项。更多信息可参考 `fstab` 的帮助（ `man fstab` ）。

参考资料：

- Linux 硬盘分区以及其挂载原理
- 从文件 I/O 看 Linux 的虚拟文件系统
- 源码分析：静态分析 C 程序函数调用关系图

如何制作一个文件系统

Linux 文件系统下有一些最基本的目录，不同的目录下存放着不同作用的各类文件。最基本的目录有 `/etc` ， `/lib` ， `/dev` ， `/bin` 等，它们分别存放着系统配置文件，库文件，设备文件和可执行程序。这些目录一般情况下是必须的，在做嵌入式开发时，需要手动或者是用 `busybox` 等工具来创建这样一个基本的文件系统。这里仅制作一个非常简单的文件系统，并对该文件系统进行各种常规操作，以便加深对文件系统的理解。

范例：用 **dd** 创建一个固定大小的文件

还记得 `dd` 命令么？就用它来产生一个固定大小的文件，这个为 `1M(1024\*1024 bytes)` 的文件

```
$ dd if=/dev/zero of=minifs bs=1024 count=1024
```

查看文件类型，这里的 `minifs` 是一个充满 `\\0` 的文件，没有任何特定的数据结构

```
$ file minifs
minifs: data
```

说明： `/dev/zero` 是一个非常特殊的设备，如果读取它，可以获取任意多个 `\\0` 。

接着把该文件格式化为某个指定文件类型的文件系统。（是不是觉得不可思议，文件也可以格式化？是的，不光是设备可以，文件也可以以某种文件系统类型进行组织，但是需要注意的是，某些文件系统（如 `ext3` ）要求被格式化的目标最少有 `64M` 的空间）。

范例：用 **mkfs** 格式化文件

```
$ mkfs.ext2 minifs
```

查看此时的文件类型，这时文件 `minifs` 就以 `ext2` 文件系统的格式组织了

```
$ file minifs
minifs: Linux rev 1.0 ext2 filesystem data
```

范例：挂载刚创建的文件系统

因为该文件以文件系统的类型组织了，那么可以用 `mount` 命令挂载并使用它。

请切换到 `root` 用户挂载它，并通过 `-o loop` 选项把它关联到一个特殊设备 `/dev/loop`

```
$ sudo -s
# mount minifs /mnt/ -o loop
```

查看该文件系统信息，仅可以看到一个目录文件 `lost+found`

```
$ ls /mnt/
lost+found
```

范例：对文件系统进行读、写、删除等操作

在该文件系统下进行各种常规操作，包括读、写、删除等。(每次操作前先把 `minifs` 文件保存一份，以便比较，结合相关资料就可以深入地分析各种操作对文件系统的改变情况，从而深入理解文件系统作为一种组织数据的方式的实现原理等)

```
$ cp minifs minifs.bak
$ cd /mnt
$ touch hello
$ cd -
$ cp minifs minifs-touch.bak
$ od -x minifs.bak > orig.od
$ od -x minifs-touch.bak > touch.od
```

创建一个文件后，比较此时文件系统和之前文件系统的异同

```
$ diff orig.od touch.od
diff orig.od touch.od
61,63c61,64
< 0060020 000c 0202 2e2e 0000 000b 0000 03e8 020a
< 0060040 6f6c 7473 662b 756f 646e 0000 0000 0000
< 0060060 0000 0000 0000 0000 0000 0000 0000 0000
---
> 0060020 000c 0202 2e2e 0000 000b 0000 0014 020a
> 0060040 6f6c 7473 662b 756f 646e 0000 000c 0000
> 0060060 03d4 0105 6568 6c6c 006f 0000 0000 0000
> 0060100 0000 0000 0000 0000 0000 0000 0000 0000
```

通过比较发现：添加文件，文件系统的相应位置发生了明显的变化

```
$ echo "hello, world" > /mnt/hello
```

执行 `sync` 命令，确保缓存中的数据已经写入磁盘（还记得本节图 1 的 `buffer cache` 吧，这里就是把 `cache` 中的数据写到磁盘中）

```
$ sync
$ cp minifs minifs-echo.bak
$ od -x minifs-echo.bak > echo.od
```

写入文件内容后，比较文件系统和之前的异同

```
$ diff touch.od echo.od
```

查看文件系统中的字符串

```
$ strings minifs
lost+found
hello
hello, world
```

删除 `hello` 文件，查看文件系统变化

```
$ rm /mnt/hello
$ cp minifs minifs-rm.bak
$ od -x minifs-rm.bak > rm.od
$ diff echo.od rm.od
```

通过查看文件系统的字符串发现：删除文件时并没有覆盖文件内容，所以从理论上说内容此时还是可恢复的

```
$ strings minifs
lost+found
hello
hello, world
```

上面仅仅演示了一些分析文件系统的常用工具，并分析了几个常规的操作，如果想非常深入地理解文件系统的实现原理，请熟悉使用上述工具并阅读相关资料。

参考资料：

- Build a mini filesystem in linux from scratch

- Build a mini filesystem in linux with BusyBox
- ext2 文件系统

如何开发自己的文件系统

随着 `fuse` 的出现，在用户空间开发文件系统成为可能，如果想开发自己的文件系统，那么推荐阅读：使用 fuse 开发自己的文件系统。

后记

- 2007 年 12 月 22 日，收集了很多资料，写了整体的框架
- 2007 年 12 月 28 日下午，完成初稿，考虑到时间关系，很多细节也没有进一步分析，另外有些部分可能存在理解上的问题，欢迎批评指正
- 2007 年 12 月 28 日晚，修改部分资料，并正式公开该篇文档
- 29 号，添加设备驱动和硬件设备一小节

进程操作

前言

进程作为程序真正发挥作用时的“形态”，我们有必要对它的一些相关操作非常熟悉，这一节主要描述进程相关的概念和操作，将介绍包括程序、进程、作业等基本概念以及进程状态查询、进程通信等相关的操作。

什么是程序，什么又是进程

程序是指令的集合，而进程则是程序执行的基本单元。为了让程序完成它的工作，必须让程序运行起来成为进程，进而利用处理器资源、内存资源，进行各种 `I/O` 操作，从而完成某项特定工作。

从这个意思上说，程序是静态的，而进程则是动态的。

进程有区别于程序的地方还有：进程除了包含程序文件中的指令数据以外，还需要在内核中有一个数据结构用以存放特定进程的相关属性，以便内核更好地管理和调度进程，从而完成多进程协作的任务。因此，从这个意义上可以说“高于”程序，超出了程序指令本身。

如果进行过多进程程序的开发，又会发现，一个程序可能创建多个进程，通过多个进程的交互完成任务。在 Linux 下，多进程的创建通常是通过 `fork` 系统调用来实现。从这个意义上来说程序则”包含”了进程。

另外一个需要明确的是，程序可以由多种不同程序语言描述，包括 C 语言程序、汇编语言程序和最后编译产生的机器指令等。

下面简单讨论 Linux 下面如何通过 Shell 进行进程的相关操作。

进程的创建

通常在命令行键入某个程序文件名以后，一个进程就被创建了。例如，

范例：让程序在后台运行

```
$ sleep 100 &
[1] 9298
```

范例：查看进程 ID

用 `pidof` 可以查看指定程序名的进程ID：

```
$ pidof sleep
9298
```

范例：查看进程的内存映像

```
$ cat /proc/9298/maps
08048000-0804b000 r-xp 00000000 08:01 977399     /bin/sleep
0804b000-0804c000 rw-p 00003000 08:01 977399     /bin/sleep
0804c000-0806d000 rw-p 0804c000 00:00 0          [heap]
b7c8b000-b7cca000 r--p 00000000 08:01 443354
...
bfbd8000-bfbed000 rw-p bfbd8000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

程序被执行后，就被加载到内存中，成为了一个进程。上面显示了该进程的内存映像（虚拟内存），包括程序指令、数据，以及一些用于存放程序命令行参数、环境变量的栈空间，用于动态内存申请的堆空间都被分配好。

关于程序在命令行执行过程的细节，请参考《Linux 命令行下程序执行的一刹那》。

实际上，创建一个进程，也就是说让程序运行，还有其他的办法，比如，通过一些配置让系统启动时自动启动程序（具体参考 `man init`），或者是通过配置 `crond` （或者 `at`）让它定时启动程序。除此之外，还有一个方式，那就是编写 Shell 脚本，把程序写入一个脚本文件，当执行脚本文件时，文件中的程序将被执行而成为进程。这些方式的细节就不介绍，下面了解如何查看进程的属性。

需要补充一点的是：在命令行下执行程序，可以通过 `ulimit` 内置命令来设置进程可以利用的资源，比如进程可以打开的最大文件描述符个数，最大的栈空间，虚拟内存空间等。具体用法见 `help ulimit` 。

查看进程的属性和状态

可以通过 `ps` 命令查看进程相关属性和状态，这些信息包括进程所属用户，进程对应的程序，进程对 `cpu` 和内存的使用情况等信息。熟悉如何查看它们有助于进行相关的统计分析等操作。

范例：通过 ps 命令查看进程属性

查看系统当前所有进程的属性：

```
$ ps -ef
```

查看命令中包含某字符的程序对应的进程，进程 `ID` 是 1 。 `TTY` 为？表示和终端没有关联：

```
$ ps -C init
  PID TTY          TIME CMD
    1 ?        00:00:01 init
```

选择某个特定用户启动的进程：

```
$ ps -U falcon
```

按照指定格式输出指定内容，下面输出命令名和 `cpu` 使用率：

```
$ ps -e -o "%C %c"
```

打印 `cpu` 使用率最高的前 4 个程序：

```
$ ps -e -o "%C %c" | sort -u -k1 -r | head -5
 7.5 firefox-bin
 1.1 Xorg
 0.8 scim-panel-gtk
 0.2 scim-bridge
```

获取使用虚拟内存最大的 5 个进程：

```
$ ps -e -o "%z %c" | sort -n -k1 -r | head -5
349588 firefox-bin
 96612 xfce4-terminal
 88840 xfdesktop
 76332 gedit
 58920 scim-panel-gtk
```

范例：通过 pstree 查看进程亲缘关系

系统所有进程之间都有“亲缘”关系，可以通过 `pstree` 查看这种关系：

```
$ pstree
```

上面会打印系统进程调用树，可以非常清楚地看到当前系统中所有活动进程之间的调用关系。

范例：用top动态查看进程信息

```
$ top
```

该命令最大特点是可以动态地查看进程信息，当然，它还提供了一些其他的参数，比如 `-S` 可以按照累计执行时间的大小排序查看，也可以通过 `-u` 查看指定用户启动的进程等。

补充： `top` 命令支持交互式，比如它支持 `u` 命令显示用户的所有进程，支持通过 `k` 命令杀掉某个进程；如果使用 `-n 1` 选项可以启用批处理模式，具体用法为：

```
$ top -n 1 -b
```

范例：确保特定程序只有一个副本在运行

下面来讨论一个有趣的问题：如何让一个程序在同一时间只有一个在运行。

这意味着当一个程序正在被执行时，它将不能再被启动。那该怎么做呢？

假如一份相同的程序被复制成了很多份，并且具有不同的文件名被放在不同的位置，这个将比较糟糕，所以考虑最简单的情况，那就是这份程序在整个系统上是唯一的，而且名字也是唯一的。这样的话，有哪些办法来回答上面的问题呢？

总的机理是：在程序开头检查自己有没有执行，如果执行了则停止否则继续执行后续代码。

策略则是多样的，由于前面的假设已经保证程序文件名和代码的唯一性，所以通过 `ps` 命令找出当前所有进程对应的程序名，逐个与自己的程序名比较，如果已经有，那么说明自己已经运行了。

```
ps -e -o "%c" | tr -d " " | grep -q ^init$   #查看当前程序是否执行
[ $? -eq 0 ] && exit   #如果在，那么退出, $?表示上一条指令是否执行成功
```

每次运行时先在指定位置检查是否存在一个保存自己进程 `ID` 的文件，如果不存在，那么继续执行，如果存在，那么查看该进程 `ID` 是否正在运行，如果在，那么退出，否则往该文件重新写入新的进程 `ID`，并继续。

```
pidfile=/tmp/$0".pid"
if [ -f $pidfile ]; then
       OLDPID=$(cat $pidfile)
    ps -e -o "%p" | tr -d " " | grep -q "^$OLDPID$"
    [ $? -eq 0 ] && exit
fi

echo $$ > $pidfile

#... 代码主体

#设置信号0的动作，当程序退出时触发该信号从而删除掉临时文件
trap "rm $pidfile"      0
```

更多实现策略自己尽情发挥吧！

调整进程的优先级

在保证每个进程都能够顺利执行外，为了让某些任务优先完成，那么系统在进行进程调度时就会采用一定的调度办法，比如常见的有按照优先级的时间片轮转的调度算法。这种情况下，可以通过 `renice` 调整正在运行的程序的优先级，例如：`

范例：获取进程优先级

```
$ ps -e -o "%p %c %n" | grep xfs
 5089 xfs               0
```

范例：调整进程的优先级

```
$ renice 1 -p 5089
renice: 5089: setpriority: Operation not permitted
$ sudo renice 1 -p 5089   #需要权限才行
[sudo] password for falcon:
5089: old priority 0, new priority 1
$ ps -e -o "%p %c %n" | grep xfs  #再看看，优先级已经被调整过来了
 5089 xfs               1
```

结束进程

既然可以通过命令行执行程序，创建进程，那么也有办法结束它。可以通过 `kill` 命令给用户自己启动的进程发送某个信号让进程终止，当然“万能”的 `root` 几乎可以 `kill` 所有进程（除了 `init` 之外）。例如，

范例：结束进程

```
$ sleep 50 &   #启动一个进程
[1] 11347
$ kill 11347
```

`kill` 命令默认会发送终止信号（ `SIGTERM` ）给程序，让程序退出，但是 `kill` 还可以发送其他信号，这些信号的定义可以通过 `man 7 signal` 查看到，也可以通过 `kill -l` 列出来。

```
$ man 7 signal
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

范例：暂停某个进程

例如，用 `kill` 命令发送 `SIGSTOP` 信号给某个程序，让它暂停，然后发送 `SIGCONT` 信号让它继续运行。

```
$ sleep 50 &
[1] 11441
$ jobs
[1]+  Running                 sleep 50 &
$ kill -s SIGSTOP 11441   #这个等同于我们对一个前台进程执行CTRL+Z操作
$ jobs
[1]+  Stopped                 sleep 50
$ kill -s SIGCONT 11441   #这个等同于之前我们使用bg %1操作让一个后台进程运行起来
$ jobs
[1]+  Running                 sleep 50 &
$ kill %1                  #在当前会话(session)下，也可以通过作业号控制进程
$ jobs
[1]+  Terminated              sleep 50
```

可见 `kill` 命令提供了非常好的功能，不过它只能根据进程的 `ID` 或者作业来控制进程，而 `pkill` 和 `killall` 提供了更多选择，它们扩展了通过程序名甚至是进程的用户名来控制进程的方法。更多用法请参考它们的手册。

范例：查看进程退出状态

当程序退出后，如何判断这个程序是正常退出还是异常退出呢？还记得 Linux 下，那个经典 `hello world` 程序吗？在代码的最后总是有条 `return 0` 语句。这个 `return 0` 实际上是让程序员来检查进程是否正常退出的。如果进程返回了一个其他的数值，那么可以肯定地说这个进程异常退出了，因为它都没有执行到 `return 0` 这条语句就退出了。

那怎么检查进程退出的状态，即那个返回的数值呢？

在 `Shell` 中，可以检查这个特殊的变量 `$?`，它存放了上一条命令执行后的退出状态。

```
$ test1
bash: test1: command not found
$ echo $?
127
$ cat ./test.c | grep hello
$ echo $?
1
$ cat ./test.c | grep hi
    printf("hi, myself!\n");
$ echo $?
0
```

貌似返回 0 成为了一个潜规则，虽然没有标准明确规定，不过当程序正常返回时，总是可以从 `$?` 中检测到 0，但是异常时，总是检测到一个非 0 值。这就告诉我们在程序的最后最好是跟上一个 `exit 0` 以便任何人都可以通过检测 `$?` 确定程序是否正常结束。如果有一天，有人偶尔用到你的程序，试图检查它的退出状态，而你却在程序的末尾莫名地返回了一个 `-1` 或者 1，那么他将会很苦恼，会怀疑他自己编写的程序到底哪个地方出了问题，检查半天却不知所措，因为他太信任你了，竟然从头至尾都没有怀疑你的编程习惯可能会与众不同！

进程通信

为便于设计和实现，通常一个大型的任务都被划分成较小的模块。不同模块之间启动后成为进程，它们之间如何通信以便交互数据，协同工作呢？在《UNIX 环境高级编程》一书中提到很多方法，诸如管道（无名管道和有名管道）、信号（ `signal` ）、报文（ `Message` ）队列（消息队列）、共享内存（ `mmap/munmap` ）、信号量（ `semaphore` ，主要是同步用，进程之间，进程的不同线程之间）、套接口（ `Socket` ，支持不同机器之间的进程通信）等，而在 Shell 中，通常直接用到的就有管道和信号等。下面主要介绍管道和信号机制在 Shell 编程时的一些用法。

范例：无名管道（pipe）

在 Linux 下，可以通过 `|` 连接两个程序，这样就可以用它来连接后一个程序的输入和前一个程序的输出，因此被形象地叫做个管道。在 C 语言中，创建无名管道非常简单方便，用 `pipe` 函数，传入一个具有两个元素的 `int` 型的数组就可以。这个数组实际上保存的是两个

文件描述符，父进程往第一个文件描述符里头写入东西后，子进程可以从第一个文件描述符中读出来。

如果用多了命令行，这个管子 `|` 应该会经常用。比如上面有个演示把 `ps` 命令的输出作为 `grep` 命令的输入：

```
$ ps -ef | grep init
```

也许会觉得这个“管子”好有魔法，竟然真地能够链接两个程序的输入和输出，它们到底是怎么实现的呢？实际上当输入这样一组命令时，当前 Shell 会进行适当的解析，把前面一个进程的输出关联到管道的输出文件描述符，把后面一个进程的输入关联到管道的输入文件描述符，这个关联过程通过输入输出重定向函数 `dup` （或者 `fcntl` ）来实现。

范例：有名管道（named pipe）

有名管道实际上是一个文件（无名管道也像一个文件，虽然关系到两个文件描述符，不过只能一边读另外一边写），不过这个文件比较特别，操作时要满足先进先出，而且，如果试图读一个没有内容的有名管道，那么就会被阻塞，同样地，如果试图往一个有名管道里写东西，而当前没有程序试图读它，也会被阻塞。下面看看效果。

```
$ mkfifo fifo_test    #通过mkfifo命令创建一个有名管道
$ echo "fewfefe" > fifo_test
#试图往fifo_test文件中写入内容，但是被阻塞，要另开一个终端继续下面的操作
$ cat fifo_test        #另开一个终端，记得，另开一个。试图读出fifo_test的内容
fewfefe
```

这里的 `echo` 和 `cat` 是两个不同的程序，在这种情况下，通过 `echo` 和 `cat` 启动的两个进程之间并没有父子关系。不过它们依然可以通过有名管道通信。

这样一种通信方式非常适合某些特定情况：例如有这样一个架构，这个架构由两个应用程序构成，其中一个通过循环不断读取 `fifo_test` 中的内容，以便判断，它下一步要做什么。如果这个管道没有内容，那么它就会被阻塞在那里，而不会因死循环而耗费资源，另外一个则作为一个控制程序不断地往 `fifo_test` 中写入一些控制信息，以便告诉之前的那个程序该做什么。下面写一个非常简单的例子。可以设计一些控制码，然后控制程序不断地往 `fifo_test` 里头写入，然后应用程序根据这些控制码完成不同的动作。当然，也可以往 `fifo_test` 传入除控制码外的其他数据。

- 应用程序的代码

```
$ cat app.sh
#!/bin/bash

FIFO=fifo_test
while :;
do
    CI=`cat $FIFO`  #CI --> Control Info
    case $CI in
        0) echo "The CONTROL number is ZERO, do something ..."
            ;;
        1) echo "The CONTROL number is ONE, do something ..."
            ;;
        *) echo "The CONTROL number not recognized, do something else..."
            ;;
    esac
done
```

- 控制程序的代码

```
$ cat control.sh
#!/bin/bash

FIFO=fifo_test
CI=$1

[ -z "$CI" ] && echo "the control info should not be empty" && exit

echo $CI > $FIFO
```

- 一个程序通过管道控制另外一个程序的工作

```
$ chmod +x app.sh control.sh    #修改这两个程序的可执行权限，以便用户可以执行它们
$ ./app.sh  #在一个终端启动这个应用程序，在通过./control.sh发送控制码以后查看输出
The CONTROL number is ONE, do something ...    #发送1以后
The CONTROL number is ZERO, do something ...    #发送0以后
The CONTROL number not recognized, do something else...  #发送一个未知的控制码以后
$ ./control.sh 1            #在另外一个终端，发送控制信息，控制应用程序的工作
$ ./control.sh 0
$ ./control.sh 4343
```

这样一种应用架构非常适合本地的多程序任务设计，如果结合 `web cgi` ，那么也将适合远程控制的要求。引入 `web cgi` 的唯一改变是，要把控制程序 `./control.sh` 放到 `web` 的 `cgi` 目录下，并对它作一些修改，以使它符合 `CGI` 的规范，这些规范包括文档输出格式的表示（在文件开头需要输出 `content-tpye: text/html` 以及一个空白行）和输入参数的获取 `(web` 输入参数都存放在 `QUERY_STRING` 环境变量里头）。因此一个非常简单的 `CGI` 控制程序可以写成这样：

```
#!/bin/bash

FIFO=./fifo_test
CI=$QUERY_STRING

[ -z "$CI" ] && echo "the control info should not be empty" && exit

echo -e "content-type: text/html\n\n"
echo $CI > $FIFO
```

在实际使用时，请确保 `control.sh` 能够访问到 `fifo_test` 管道，并且有写权限，以便通过浏览器控制 `app.sh` ：

```
http://ipaddress\_or\_dns/cgi-bin/control.sh?0
```

问号 `?` 后面的内容即 `QUERY_STRING`，类似之前的 `$1` 。

这样一种应用对于远程控制，特别是嵌入式系统的远程控制很有实际意义。在去年的暑期课程上，我们就通过这样一种方式来实现马达的远程控制。首先，实现了一个简单的应用程序以便控制马达的转动，包括转速，方向等的控制。为了实现远程控制，我们设计了一些控制码，以便控制马达转动相关的不同属性。

在 C 语言中，如果要使用有名管道，和 Shell 类似，只不过在读写数据时用 `read`，`write` 调用，在创建 `fifo` 时用 `mkfifo` 函数调用。

范例：信号（Signal）

信号是软件中断，Linux 用户可以通过 `kill` 命令给某个进程发送一个特定的信号，也可以通过键盘发送一些信号，比如 `CTRL+C` 可能触发 `SGIINT` 信号，而 `CTRL+\` 可能触发 `SGIQUIT` 信号等，除此之外，内核在某些情况下也会给进程发送信号，比如在访问内存越界时产生 `SGISEGV` 信号，当然，进程本身也可以通过 `kill`，`raise` 等函数给自己发送信号。对于 Linux 下支持的信号类型，大家可以通过 `man 7 signal` 或者 `kill -l` 查看到相关列表和说明。

对于有些信号，进程会有默认的响应动作，而有些信号，进程可能直接会忽略，当然，用户还可以对某些信号设定专门的处理函数。在 Shell 中，可以通过 `trap` 命令（Shell 内置命令）来设定响应某个信号的动作（某个命令或者定义的某个函数），而在 C 语言中可以通过 `signal` 调用注册某个信号的处理函数。这里仅仅演示 `trap` 命令的用法。

```
$ function signal_handler { echo "hello, world."; } #定义signal_handler函数
$ trap signal_handler SIGINT  #执行该命令设定：收到SIGINT信号时打印hello, world
$ hello, world     #按下CTRL+C，可以看到屏幕上输出了hello, world字符串
```

类似地，如果设定信号 0 的响应动作，那么就可以用 `trap` 来模拟 C 语言程序中的 `atexit` 程序终止函数的登记，即通过 `trap signal_handler SIGQUIT` 设定的 `signal_handler` 函数将在程序退出时执行。信号 0 是一个特别的信号，在 `POSIX.1` 中把信号编号 0 定义为空信号，这常被用来确定一个特定进程是否仍旧存在。当一个程序退出时会触发该信号。

```
$ cat sigexit.sh
#!/bin/bash

function signal_handler {
    echo "hello, world"
}
trap signal_handler 0
$ chmod +x sigexit.sh
$ ./sigexit.sh    #实际Shell编程会用该方式在程序退出时来做一些清理临时文件的收尾工作
hello, world
```

作业和作业控制

当我们为完成一些复杂的任务而将多个命令通过 `|,\>,<, ;, (,)` 等组合在一起时，通常这个命令序列会启动多个进程，它们间通过管道等进行通信。而有时在执行一个任务的同时，还有其他的任务需要处理，那么就经常会在命令序列的最后加上一个&，或者在执行命令后，按下 `CTRL+Z` 让前一个命令暂停。以便做其他的任务。等做完其他一些任务以后，再通过 `fg` 命令把后台任务切换到前台。这样一种控制过程通常被成为作业控制，而那些命令序列则被成为作业，这个作业可能涉及一个或者多个程序，一个或者多个进程。下面演示一下几个常用的作业控制操作。

范例：创建后台进程，获取进程的作业号和进程号

```
$ sleep 50 &
[1] 11137
```

范例：把作业调到前台并暂停

使用 Shell 内置命令 `fg` 把作业 1 调到前台运行，然后按下 `CTRL+Z` 让该进程暂停

```
$ fg %1
sleep 50
^Z
[1]+  Stopped                 sleep 50
```

范例：查看当前作业情况

```
$ jobs              #查看当前作业情况，有一个作业停止
[1]+  Stopped                 sleep 50
$ sleep 100 &       #让另外一个作业在后台运行
[2] 11138
$ jobs              #查看当前作业情况，一个正在运行，一个停止
[1]+  Stopped                 sleep 50
[2]-  Running                 sleep 100 &
```

范例：启动停止的进程并运行在后台

```
$ bg %1
[2]+ sleep 50 &
```

不过，要在命令行下使用作业控制，需要当前 Shell，内核终端驱动等对作业控制支持才行。

参考资料

- 《UNIX 环境高级编程》

网络操作

前言

前面章节已经介绍了Shell编程范例之数值、布尔值、字符串、文件、文件系统、进程等的操作。这些内容基本覆盖了网络中某个独立机器正常工作的“方方面面”，现在需要把视角从单一的机器延伸到这些机器通过各种网络设备和协议连接起来的网络世界，分析网络拓扑结构、网络工作原理、了解各种常见网络协议、各种常见硬件工作原理、网络通信与安全相关软件以及工作原理分析等。

不过，因为网络相关的问题确实太复杂了，这里不可能介绍具体，因此如果想了解更多细节，还是建议参考相关资料。但Linux是一个网络原理学习和实践的好平台，不仅因为它本身对网络体系结构的实现是开放源代码的，而且各种相关的分析工具和函数库数不胜数，因此，如果你是学生，千万不要错过通过它来做相关的实践工作。

网络原理介绍

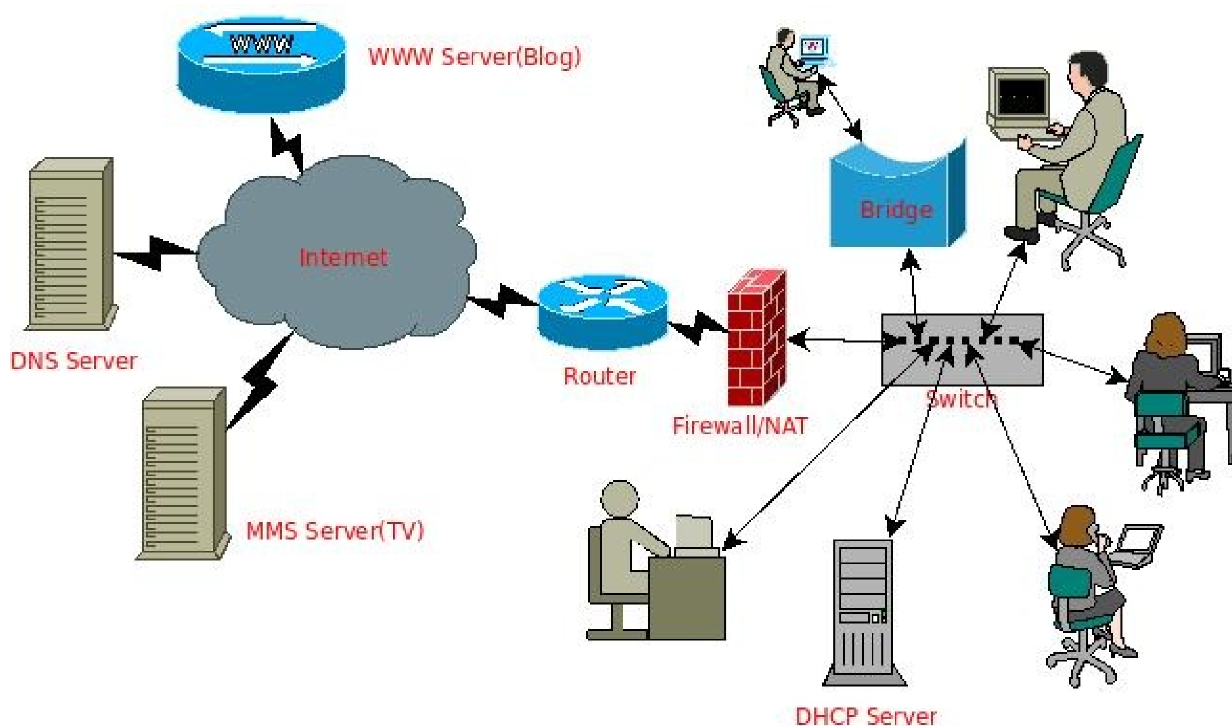
我们的网络世界

在进行所有介绍之前，来直观地感受一下那个真真实实存在的网络世界吧。当我在 Linux 下通过 `Web` 编辑器写这篇 Blog 时，一边用 `mplayer` 听着远程音乐，累了时则打开兰大的网络 `TV` 频道开始看看凤凰卫视……这些“现代化”的生活，我想，如果没有网络，将变得无法想象。

下面来构想一下这样一个网络世界的优美图画：

> 一边盯着显示器，一边敲击着键盘，一边挂着耳机。
>
> 主机电源灯灿烂得很，发着绿光，这时很容易想象主机背后的那个网卡位置肯定有两个不同颜色的灯光在闪烁，它显示着主机正在与计算机网络世界打着交道。
>
> 就在实验室的某个角落，有一个交换机上的一个网口的网线连到主机上，这个交换机接到了一个局域网的网关上，然后这个网关再接到了信息楼的某个路由器上，再转接到学校网络中心的另外一个路由器上……
>
> 期间，有一个路由器连接到了这个 Blog 服务器上，而另外一个则可能连到了那个网络 `TV` 服务器上，还有呢，另外一些则连接到了电信网络里头的某个音乐服务器上……

下面用 `dia` 绘制一个简单的“网络地图”：



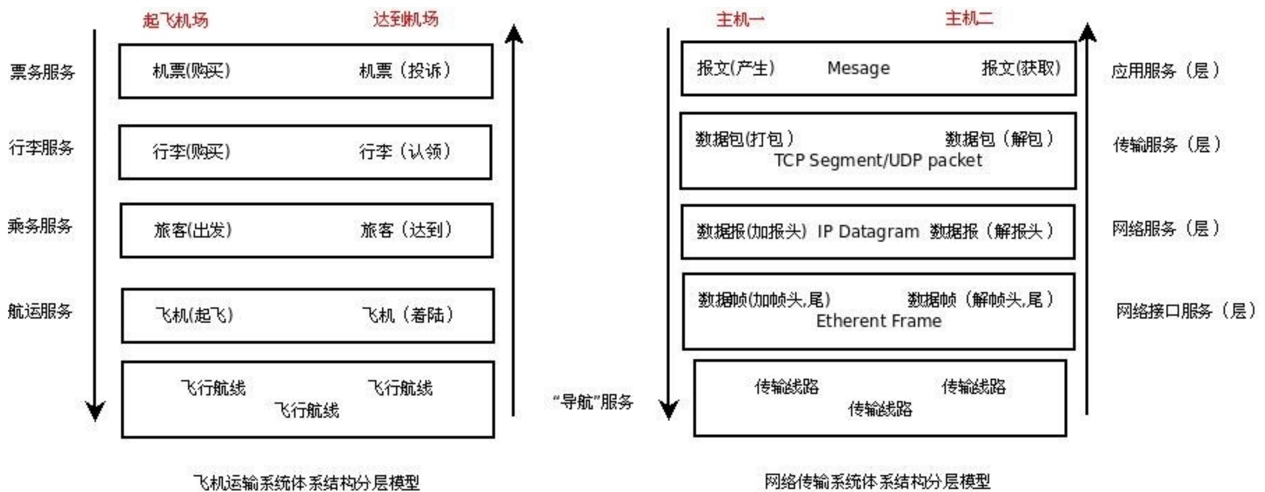
该图把一些最常见的网络设备和网络服务基本都呈现出来了，包括本地主机、路由、交换机、网桥，域名服务器，万维网服务，视频服务，防火墙服务，动态 `IP` 地址服务等。其中各种设备构成了整个物理网络，而网络服务则是构建在这些设备上的各种网络应用。

现在的网络应用越来越丰富多样，比如即时聊天（ `IM` ）、 `p2p` 资源共享、网络搜索等，它们是如何实现的，它们如何构建在各种各样的网络设备之上，并且能够安全有效的工作呢？这取决于这背后逐步完善的网络体系结构和各种相关网络协议的开发、实现和应用。

网络体系结构和网络协议介绍

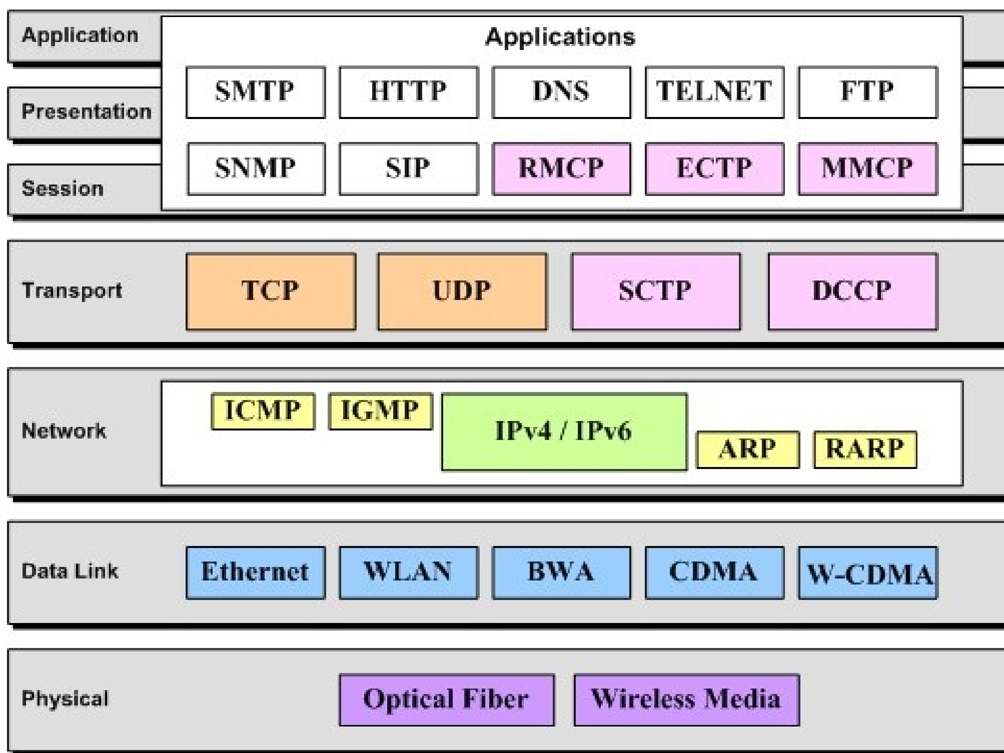
那么网络体系结构是怎么样的呢？涉及到哪些相关的网络协议呢？什么又是网络协议呢？

在《计算机网络——自顶向下的方法》一书中非常巧妙地给出了网络体系结构分层的比喻，把网络中各层跟交通运输体系中的各个环节对照起来，让人通俗易懂。在交通运输体系中，运输的是人和物品，在计算机网络体系中，运输的是电子数据。考虑到交通运输网络和计算机网络中最终都可以划归为点对点的信息传输。这里考虑两点之间的信息传递过程，得到这样一个对照关系，见下图：

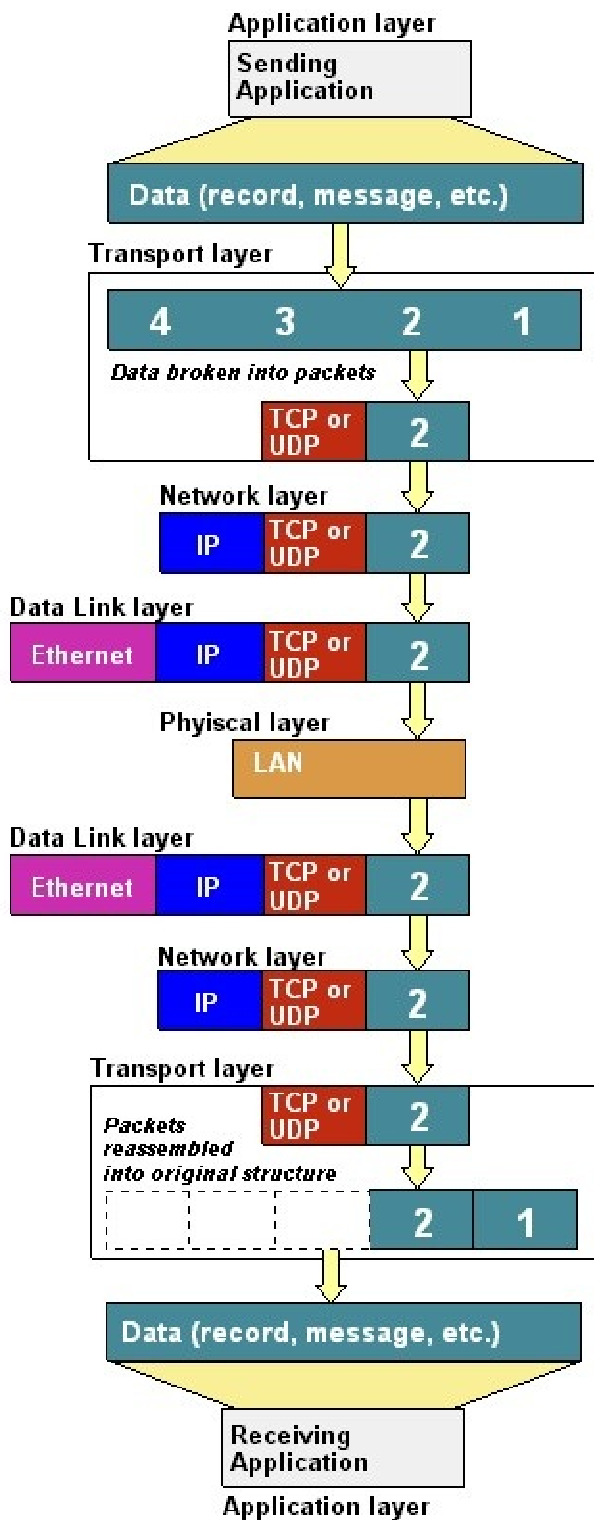


对照上图，更容易理解右侧网络体系结构的分层原理（如果比照一封信发出到收到的这一中间过程可能更容易理解），上图右侧是 `TCP/IP` 网络体系结构的一个网络分层示意图，在把数据发送到网络之前，在各层中需要进行各种“打包”的操作，而从网络接收到数据后，就需要进行“解包”操作，最终把纯粹的数据信息给提取出来。这种分层的方式是为了传输数据的需要，也是两个主机之间如何建立连接以及如何保证数据传输的完整性和可靠性的需要。通过把各种需要分散在不同的层次，使得整个体系结构更加清晰和明了。这些“需求”具体通过各种对应的协议来规范，这些规范统成为网络协议。

关于 `OSI` 模型（7 层）比照 `TCP/IP` 模型（4 层）的协议栈可以从下图（来自网络）看个明了：



而下图（来自网络）则更清晰地体现了 `TCP/IP` 分层模型。



上面介绍了网络原理方面的基本内容，如果想了解更多网络原理和操作系统对网络支持的实现，可以考虑阅读后面的参考资料。下面将做一些实践，即在 Linux 下如何联网，如何用 Linux 搭建各种网络服务，并进行网络安全方面的考量以及基本的网络编程和开发的介绍。

Linux 下网络“实战”

如何把我们的 Linux 主机接入网络

如果要让一个系统能够联网，首先当然是搭建好物理网络了。接入网络的物理方式还是蛮多的，比如直接用网线接入以太网，用无线网卡上网，用 `ADSL` 拨号上网......

对于用以太网网卡接入网络的常见方式，在搭建好物理网络并确保连接正常后，可以通过配置 `IP` 地址和默认网关来接入网络，这个可以通过手工配置和动态获取两种方式。

范例：通过dhclient获取IP地址

如果所在的局域网有 `DHCP` 服务，那么可以这么获取，`N` 是设备名称，如果只有一块网卡，一般是 0 或者 1 。

```
$ dhclient ethN
```

范例：静态配置IP地址

当然，也可以考虑采用静态配置的方式，`ip_address` 是本地主机的 `IP` 地址，`gw_ip_address` 是接入网络的网关的 `IP` 地址。

```
$ ifconfig eth0 ip_address on
$ route add deafult gw gw_ip_address
```

如果上面不工作，记得通过 `ifconfig/mii-tool/ethtool` 等工具检查网卡是否有被驱动起来，然后通过 `lspci/dmesg` 等检查网卡类型（或者通过主板手册和独立网卡自带的手册查看），接着安装或者编译相关驱动，最后把驱动通过 `insmod/modprobe` 等工具加载到内核中。

用 Linux 搭建网桥

网桥工作在 `OSI` 模型的第二层，即数据链路层，它只需要知道目标主机的 `MAC` 地址就可以工作。Linux 内核在 `2.2` 开始就已经支持了这个功能，具体怎么配置看看后续参考资料吧。如果要把 Linux 主机配置成一个网桥，至少需要两个网卡。

网桥的作用相当于一根网线，用户无须关心里头有什么东西，把它的两个网口连接到两个主机上就可以让这两个主机支持相互通信。不过它比网线更厉害，如果配上防火墙，就可以隔离连接在它两端的网段（注意这里是网络，因为它不识别 `IP` ），另外，如果这个网桥有多个网口，那么可以实现一个功能复杂的交换机，而如果有效组合多个网桥，则有可能实现一个复杂的可实现流量控制和负载平衡的防火墙系统。

用 Linux 做路由

路由工作在 `OSI` 模型的第三层，即网络层，通过 `router` 可以配置 Linux 的路由，当然，Linux 下也有很多工具支持动态路由的。相关的资料在网路中铺天盖地，由于时间关系，这里不做介绍。

用 Linux 搭建各种常规的网络服务

需要什么网络服务呢？

- 给局域网弄个 `DHCP` 服务器，那就弄个 `dhcpd`，看看参考资料；
- 如果想弄个邮件发送服务器，那就安装个 `sendmail` 或者 `exim4` ；
- 如果再想弄个邮件列表服务器呢，那就装个 `mailman` ；
- 如果想弄个接收邮件的服务器呢，那就安装个 `pop3` 服务器；
- 如果想弄个 `web` 站点，那就弄个 `apache` 或者 `nginx` 服务器；
- 如果想弄上防火墙服务，那么通过 `iptables` 工具配置 `netfilter` 就可以

What's more？如果你能想到，Linux上基本都有相应的实现。

Linux 下网络问题诊断与维护

如果出现网络问题，不要惊慌，逐步检查网络的各个层次：物理链接、链路层、网络层直到应用层，熟悉使用各种如下的工具，包括 `ethereal/tcpdump`，`hping`，`nmap`，`netstat`，`netpipe`，`netperf`，`vnstat`，`ntop` 等。

关于这些工具的详细用法和网络问题诊断和维护的相关知识，请看后续相关资料。

Linux 下网络编程与开发

如果想做网络编程开发，比如：

- 要实现一个客户端 `/` 服务器架构的应用，可以采用 Linux 下的 `socket` 编程了；
- 如果想写一个数据包抓获和协议分析的程序，可以采用 `libpap` 等函数库；
- 如果想实现某个协议呢，那就可以参考相关的 `RFC` 文档，并通过 `socket` 编程来实现。

这个可以参考相关的 `Linux socket` 编程等资料。

后记

本来介绍网络相关的一些基本内容，但因时间关系，没有详述，更多细节请参考相关资料。

到这里，整个《Shell编程范例》算是很粗略地完成了，不过“范例”却缺少实例，特别是这一节。因此，如果时间允许，会逐步补充一些实例。

参考资料

- 计算机网络——自上而下的分析方法
- Linux 网络体系结构（清华大学出版社出版）
- Linux 系统故障诊断与排除 第13章 网络问题（人民邮电出版社）
- 在 Linux 下用 ADSL 拨号上网
- Linux 下无线网络相关资料收集
- Linux网桥的实现分析与使用
- DHCP mini howto
- 最佳的 75 个安全工具
- 网络管理员必须掌握的知识
- Linux 上检测 rootkit 的两种工具: Rootkit Hunter 和 Chkrootkit
- 数据包抓获与 ip 协议的简单分析（基于 pcap 库）
- RFC
- HTTP 协议的 C 语言编程实现实例

用户管理

在初次撰写本书时，都只讨论到了“物”，而没有关注“人”。而在实际使用中，Linux 系统首先是面向用户的系统，所有之前介绍的内容全部是提供给不同的用户使用的。实际使用中常常碰到各类用户操作，所以这里添加一个独立的章节来介绍。

Linux 支持多用户，也就是说允许不同的人使用同一个系统，每个人有一个属于自己的帐号。而且允许大家设置不同的认证密码，确保大家的私有信息得到保护。另外，为了确保整个系统的安全，用户权限又做了进一步划分，包括普通用户和系统管理员。普通用户只允许访问自己账户授权下的信息，而系统管理员才能访问所有资源。普通用户如果想行使管理员的职能，必须获得系统管理员的许可。

为避免分散注意力，咱们不去介绍背后的那些数据文件：`/etc/passwd`，`/etc/shadow`，`/etc/group`，`/etc/gshadow`

如果确实有需要，大家可通过如下命令查看帮助：`man 5 passwd`，`man shadow`，`man group` 和 `man gshadow`

下面我们分如下几个部分来介绍：

- 用户帐号
- 用户口令
- 用户组别
- 用户和组
- 用户切换

用户帐号

帐号操作主要是增、删、改、禁。Linux 系统提供了底层的 `useradd`, `userdel` 和 `usermod` 来完成相关操作，也提供了进一步的简化封装：`adduser`, `deluser`。为了避免混淆，咱们这里只介绍最底层的指令，这些指令设计上已经够简洁明了方便。

由于只有系统管理员才能创建新用户，请确保以 root 帐号登录或者可以通过 sudo 切换为管理员帐号。

添加

创建家目录并指定登录 Shell：

```
# useradd -s /bin/bash -m test
# groups test
test : test
```

并加入所属组：

```
# useradd -s /bin/bash -m -G docker test
# groups test
test : test docker
```

删除

删除用户以及家目录等：

```
# userdel -r test
```

修改

常常用来修改默认的 Shell：

```
# usermod -s /bin/bash test
```

或者把用户加入某个新安装软件所属的组：

```
# usermod -a -G docker test
```

修改登录用户名并搬到新家：

```
# usermod -d /home/new_test -m -l new_test test
```

禁用

如果想禁用某个帐号：

```
# usermod -L test
# usermod --expiredate 1 test
```

用户口令

口令操作主要是设置、删除、修改和禁用。Linux 系统提供了 `passwd` 命令来管理用户口令。

设置

设置用户 test 的初始密码：

```
$ passwd test
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

删除

让用户 test 无须密码登录（密码为空）：

```
$ passwd -d test
```

这个很方便某些安全无关紧要的条件下（比如已登录主机中的虚拟机），可避免每次频繁输入密码。

修改

```
$ passwd test
Changing password for test.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

禁用

禁止用户通过密码登录：

```
$ passwd -l user
```

为了安全起见或者为了避免暴力破解，我们通常可以禁用密码登录，而只允许通过 SSH Key 登录。

如果要真地禁用整个帐号的使用，需要用上一节提到的 `usermod --expiredate 1` 。

用户组别

类似帐号，主要操作也是增、删、改。

Linux 系统提供了底层的 `groupadd` , `groupdel` 和 `groupmod` 来完成相关操作，也提供了进一步的简化封装： `addgroup` , `delgroup` 。

用户组别通常用来管理不同的资源，确保只有某个组别的用户才可以访问某类资源。当然，实际案例中，有些软件也为自己定义一个组别，只有该组别的用户才能访问该软件的一些功能。

添加

添加一个新组别：

```
# groupadd test
```

删除

```
# groupdel test
```

修改

修改组别名：

```
# groupmod -n new_test test
```

用户和组

用户和组别不能独立存在，`gpasswd` 可以用来处理两者的关系。

增加

从 docker 组中增加用户 test（等同于把 test 增加到 docker 组中）：

```
# gpasswd -a test docker

或

# usermod -a -G docker test
```

删除

从 test 组中删除用户 test：

```
# gpasswd -d test test
```

用户切换

由于支持多用户，那么在登录一个帐号后，可能需要切换到另外一个帐号下，可以通过 `su` 命令完成，而 `sudo` 则可以用来作为另外一个用户来执行命令。

切换帐号

切换到 Root 并启用 Bash：

```
$ su -s /bin/bash -
root@falcon-desktop:~#

或者

$ sudo -s
```

切换到普通用户：

```
$ su -s /bin/bash - test
test@falcon-desktop:~$

或者

$ sudo -i -u test
test@falcon-desktop:~$
```

免密码切到 Root

首先得把用户加入到 sudo 用户组：

```
# usermod -a -G sudo falcon
```

否则，会看到如下信息：

```
$ sudo -s
[sudo] password for test:
test is not in the sudoers file.  This incident will be reported.
```

加入 sudo 用户组以后：

```
$ sudo -s
[sudo] password for test:
```

要实现免密切换，需要先修改 `/etc/sudoers` ，加入如下一行：

```
test ALL=(ALL) NOPASSWD: ALL
```

或者在 `/etc/sudoers.d/` 下创建一个文件并加入上述内容。

```
# echo "test ALL=(ALL) NOPASSWD: ALL" > /etc/sudoers.d/test
# chmod 440 /etc/sudoers.d/test
```

总结

前言

到这里，整个 Shell 编程系列就要结束了，作为总结篇，主要回顾一下各个小节的主要内容，并总结出 Shell 编程的一些常用框架和相关注意事项等。

Shell 编程范例回顾

TODO：主要回顾各小节的内容。

常用 Shell 编程“框架”

TODO：通过分析一些实例总结各种常见问题的解决办法，比如如何保证同一时刻每个程序只有一个运行实体（进程）。

程序优化技巧

TODO：多思考，总会有更简洁和高效的方式。

其他注意事项

TODO：比如小心 `rm -rf` 的用法，如何查看系统帮助等。

正确使用 `source` 和 `.`

仅使用 `source` 和 `.` 来执行你的环境配置等功能，建议不要用于其它用途。 在Shell中使用脚本时，使用 `bash your_script.sh` 而不是 `source your_script.sh` 或 `. your_script.sh` 。

当使用 `bash` 的时候，当前的Shell会创建一个新的子进程执行你的脚本；当使用 `source` 和 `.` 时，当前的Shell会直接解释执行 `your_script.sh` 中的代码。如果 `your_script.sh` 中包含了类似 `exit 0` 这样的代码，使用 `source` 和 `.` 执行会导致当前Shell意外地退出。

附录

Shell编程学习笔记

前言

这是作者早期的 Shell 编程学习笔记，主要包括 Shell 概述、 Shell 变量、位置参数、特殊符号、别名、各种控制语句、函数等 Shell 编程知识。

要想系统地学 Shell，应该找些较系统的资料，例如：《Shell 编程范例》和《鸟哥学习Shell Scripts》。

执行 Shell 脚本的方式

范例：输入重定向到 Bash

```
$ bash < ex1
```

可以读入 `ex1` 中的程序，并执行

范例：以脚本名作为参数

其一般形式是：

```
$ bash 脚本名 〔参数]
```

例如：

```
$ bash ex2 /usr/meng /usr/zhang
```

其执行过程与上一种方式一样，但这种方式的好处是能在脚本名后面带有参数，从而将参数值传递给程序中的命令，使一个 Shell 脚本可以处理多种情况，就如同函数调用时可根据具体问题传递相应的实参。

范例：以 . 来执行

如果以当前 Shell （以 `.` 表示）执行一个 Shell 脚本，则可以使用如下简便形式：

```
$ · ex3〔参数〕
```

范例：直接执行

将 Shell 脚本的权限设置为可执行，然后在提示符下直接执行它。

具体办法：

```
$ chmod a+x ex4
$ ./ex4
```

这个要求在 Shell 脚本的开头指明执行该脚本的具体 Shell，例如 `/bin/bash` ：

```
#!/bin/bash
```

Shell 的执行原理

Shell 接收用户输入的命令（脚本名），并进行分析。如果文件被标记为可执行，但不是被编译过的程序，Shell 就认为它是一个 Shell 脚本。 Shell 将读取其中的内容，并加以解释执行。所以，从用户的观点看，执行 Shell 脚本的方式与执行一般的可执行文件的方式相似。

因此，用户开发的 Shell 脚本可以驻留在命令搜索路径的目录之下（通常是 `/bin` 、 `/usr/bin` 等），像普通命令一样使用。这样，也就开发出自己的新命令。如果打算反复使用编好的 Shell 脚本，那么采用这种方式就比较方便。

变量赋值

可以将一个命令的执行结果赋值给变量。有两种形式的命令替换：一种是使用倒引号引用命令，其一般形式是： `命令表` 。

范例：获取当前的工作目录并存放到变量中

例如：将当前工作目录的全路径名存放到变量dir中，输入以下命令行：

```
$ dir=`pwd`
```

另一种形式是： `$(命令表)` 。上面的命令行也可以改写为：

```
$ dir=$(pwd)
```

数组

`Bash` 只提供一维数组，并且没有限定数组的大小。类似与 C 语言，数组元素的下标由 0 开始编号。获取数组中的元素要利用下标。下标可以是整数或算术表达式，其值应大于或等于 0 。用户可以使用赋值语句对数组变量赋值。

范例：对数组元素赋值

对数组元素赋值的一般形式是： `数组名[下标]=值` ，例如：

```
$ city[0]=Beijing
$ city[1]=Shanghai
$ city[2]=Tianjin
```

也可以用 `declare` 命令显式声明一个数组，一般形式是：

```
$ declare -a 数组名
```

范例：访问某个数组元素

读取数组元素值的一般格式是： `${数组名[下标]}` ，例如：

```
$ echo ${city[0]}
Beijing
```

范例：数组组合赋值

一个数组的各个元素可以利用上述方式一个元素一个元素地赋值，也可以组合赋值。定义一个数组并为其赋初值的一般形式是：

```
数组名=(值1 值2 ... 值n)
```

其中，各个值之间以空格分开。例如：

```
$ A=(this is an example of shell script)
$ echo ${A[0]} ${A[2]} ${A[3]} ${A[6]}
this an example script
$ echo ${A[8]}
```

由于值表中初值共有 7 个，所以 `A` 的元素个数也是 7 。 `A[8]` 超出了已赋值的数组 `A` 的范围，就认为它是一个新元素，由于预先没有赋值，所以它的值是空串。

若没有给出数组元素的下标，则数组名表示下标为 0 的数组元素，如 `city` 就等价于 `city[0]` 。

范例：列出数组中所有内容

使用 `*` 或 `@` 做下标，则会以数组中所有元素取代。

```
$ echo ${A[*]}
this is an example of shell script
```

范例：获取数组元素个数

```
$ echo ${#A[*]}
7
```

参数传递

假如要编写一个 Shell 来求两个数的和，可以怎么实现呢？为了介绍参数传递的用法，编写这样一个脚本：

```
$ cat > add
let sum=$1+$2
echo $sum
```

保存后，执行一下：

```
$ chmod a+x ./add
$ ./add 5 10
15
```

可以看出 5 和 10 分别传给了 `$1` 和 `$2`，这是 Shell 自己预设的参数顺序，其实也可以先定义好变量，然后传递进去。

例如，修改上述脚本得到：

```
let sum=$X+$Y
echo $sum
```

再次执行：

```
$ X=5 Y=10 ./add
15
```

可以发现，同样可以得到正确结果。

设置环境变量

export一个环境变量：

```
$ export opid=True
```

这样子就可以，如果要登陆后都生效，可以直接添加到 `/etc/profile` 或者 `~/.bashrc` 里头。

键盘读起变量值

可以通过 `read` 来读取变量值，例如，来等待用户输入一个值并且显示出来：

```
$ read -p "请输入一个值 ： "  input ; echo "你输入了一个值为 ：" $input
请输入一个值 ： 21500
你输入了一个值为 ： 21500
```

设置变量的只读属性

有些重要的 Shell 变量，赋值后不应该修改，那么可设置它为 `readonly` ：

```
$ oracle_home=/usr/oracle7/bin
$ readonly oracle_home
```

条件测试命令 test

语法： `test 表达式` 如果表达式为真，则返回真，否则，返回假。

范例：数值比较

先给出数值比较时常见的比较符：

> `-eg =；-ne !=；-gt >；-ge >=；-lt <；-le <=`

```
$ test var1 -gt var2
```

范例：测试文件属性

文件的可读、可写、可执行，是否为普通文件，是否为目录分别对应：

> `-r; -w; -x; -f; -d`

```
$ test -r filename
```

范例：字符传属性以及比较

> 串的长度为零：`-z`；非零：`-n`，如:

```
$ test -z s1
```

如果串 `s1` 长度为零，返回眞。

范例：串比较

> 相等`"s1"="s2"`；不相等 `"s1"!="s2"`

还有一种比较串的方法（可以按字典序来比较）：

```
$ if [[ 'abcde' < 'abcdf' ]]; then  echo "yeah,果然是诶"; fi
yeah,果然是诶
```

整数算术或关系运算 expr

可用该命令进行的运算有：

> 算术运算：`+ - * / %`；逻辑运算`: = ! < <= > >=`

如:

```
$ i=5;expr $i+5
```

另外，`bc` 是一个命令行计算器，可以进行一些算术计算。

控制执行流程命令

范例：条件分支命令 if

`if` 命令举例：如果第一个参数是一个普通文件名，那么分页打印该文件；否则，如果它为目录名，则进入该目录并打印该目录下的所有文件，如果也不是目录，那么提示相关信息。

```
if test -f $1
then
    pr $1>/dev/lp0
elif
    test-d $1
then
    (cd $1;pr *>/dev/lp0)
else
    echo $1 is neither a file nor a directory
fi
```

范例：case 命令举例

`case` 命令是一个基于模式匹配的多路分支命令，下面将根据用户键盘输入情况决定下一步将执行那一组命令。

```
while 〔 $reply!="y" 〕 && [ $reply!="Y" ]                        #下面将学习的循环语句
do
    echo "\nAre you want to continue?(Y/N)\c"
    read reply              #读取键盘
    case $replay in
        (y|Y) break;;          #退出循环
        (n|N) echo "\n\nTerminating\n"
              exit 0;;
            *) echo "\n\nPlease answer y or n"
            continue;       #直接返回内层循环开始出继续
    esac
done
```

范例：循环语句 while, until

语法：

```
while/until 命令表1
do
    命令表2
done
```

区别是，前者执行命令表 1 后，如果退出状态为零，那么执行 `do` 后面的命令表 2，然后回到起始处，而后者执行命令表 1 后，如果退出状态非零，才执行类似操作。例子同上。

范例：有限循环命令 for

语法：

```
for 变量名 in 字符串表
do
    命令表
done
```

举例：

```
FILE="test1.c myfile1.f pccn.h"
for i in $FILE
do
    cd ./tmp
    cp $i $i.old
    echo "$i copied"
done
```

函数

现在来看看 Shell 里头的函数用法，先看个例子：写一个函数，然后调用它显示 `Hello, World!`

```
$ cat > show
# 函数定义
function show
{
    echo $1$2;
}
H="Hello,"
W="World!"
# 调用函数，并传给两个参数H和W
show $H $W
```

演示：

```
$ chmod 770 show
$./show
Hello,World!
```

看出什么蹊跷了吗？

```
$ show $H $W
```

咱们可以直接在函数名后面跟实参。

实参顺序对应“虚参”的 `$1,$2,$3` ......

注意：假如要传入一个参数，如果这个参数中间带空格，怎么办？ 先试试看。

来显示 `Hello World` （两个单词之间有个空格）

```
function show
{
    echo $1
}
HW="Hello World"
show "$HW"
```

如果直接 `show $HW`，肯定不行，因为 `$1` 只接受到了 `Hello`，所以结果只显示 `Hello`，原因是字符串变量必须用 `"` 包含起来。

后记

感兴趣的话继续学习吧！

还有好多强大的东西等着呢，比如 `cut`，`expr`，`sed`，`awk` 等等。